(4)

# The Design and Performance of the Rollback Chip: Hardware Support for Time Warp





by

Jya-Jang Tsai

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science



Department of Computer Science

The University of Utah

October 1988

# ABSTRACT

The Time Warp mechanism offers an elegant approach to attacking difficult clock synchronization problems that arise in applications such as parallel discrete event simulation. However, because Time Warp relies on a lookahead and rollback mechanism to achieve widespread exploitation of parallelism, the state of each process must periodically be saved. Existing approaches to implementing state saving and rollback are not appropriate for large Time Warp programs. A component called the Rollback Chip (RBC) is proposed in this thesis to efficiently implement these functions. Such a component could be used in a programmable, special purpose paralle discrete event simulation engine based on Time Warp. The algorithms implemented by the rollback chip are described, as well as mechanisms that allow efficient implementation. Results of simulation studies are presented. These results show that the rollback can virtually eliminate the state saving and rollback overheads that plague current software implementations of Time Warp.

# CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# ACKNOWLEDGMENTS

I would like to express my deepest appreciation to all of the people who have contributed to this thesis. In particular, to my advisor, Richard Fujimoto, for numerous patient discussions and constructive advice. I am also thankful to Gary Lindstrom and Ganesh Gopalakrishnan, who pointed me to many interesting issues about this work. Finally, to my parents I thank them with my greatest gratitute. since without their everlasting love and unfailing faith I would not have remained here and pursued my goals.

# CHAPTER 1

# INTRODUCTION

Computer simulation is a comparatively recent member of the family of modelling techniques. However, it is perhaps the most widely used method today. A classification tree of simulation methods is shown in figure 1.1.

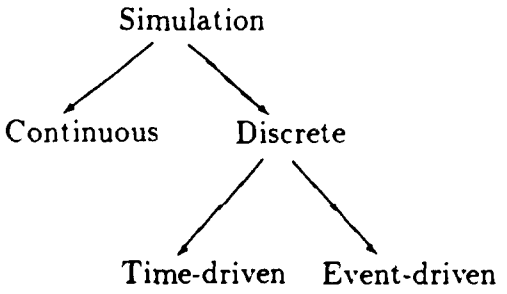



Figure 1.1. Simulation taxonomy

## 1.1 Taxonomy of Simulation Models

Divergent simulation methods have been developed for different applications. An assembly line calls for a model that is quite different from one used for weather prediction. Though divergent, simulation models can be classified into reasonably well-defined categories. Some background of the terminology commonly used in simulation is required for further discussion.

Each component of the system being simulated is represented in the simulation model by an *entity*. Each entity has zero or more *attributes* that describe its current state. The collection of all attributes is defined to be the *system state*. The state of each entity can be changed as the result of processing an *event*. The instants of time at which events occur are referred to as *event times*. For example, a simulator that models an airport might have terminals, departure flights, arriving flights etc. as its entities. For departing flights, the attributes of interest might be the flight number, the pilot, the scheduled departure time, the boarding gate, passenger list, destination, etc. The departure of a flight changes the state of the airport, so it is an event, and the actual departure time is the corresponding event time. The following subsections will describe the most commonly used taxonomy of simulation methods.

### 1.1.1 Continuous vs. discrete

A "continuous" model is one whose state varies (at least conceptually) continuously with time, e.g. fluid flow models. Such systems are usually described by sets of differential equations with associated initial and boundary conditions. Simulation models of this kind are heavily used in problems of mechanics, electrical engineering, and economics. The intervals between events are infinitesimal.

On the other hand, the state of a "discrete" model varies at discrete points in time. A model that simulates the interactions among customers and tellers in a bank service or the airport example described above are typical examples. Discrete simulation can be further divided into two sub-categories: *time-driven* and *event-driven*.

### 1.1.2 Time-driven vs. event-driven

Events in time-driven simulations may only occur at pre-defined time steps. All events that are designated to occur at the same simulated time are processed before

the simulation can proceed to events at later time steps. This method is suitable for synchronous system where many events occur simultaneously (or modelling as such does not significantly degrade accuracy), e.g. gate-level logic simulation.

Unlike time-driven, events in event-driven simulation may occur at any point in simulated time. A convenient uniprocessor implementation of a discrete-event simulation is to use an *event-list*. Events in the event list are sorted according to their event times (which is also their execution order). The simulator repeatedly removes the next event (the event with the smallest event time) from the event list, advances its clock to the simulated time of that event, and then processes the event.

The remainder of this thesis will deal exclusively in the realm of discrete-event simulations.

## 1.2 Techniques to Speed Up Simulation

Computer simulation of large, complex systems remains a major stumbling block in many research and development efforts today. Computation requirements continue to grow and far exceed the capabilities of general purpose computing hardware. The enormous amounts of computing time required to simulate large communication networks, parallel computer architectures, and battlefield scenarios (to name a few) hinder advances in systems design and development. In many cases, complex simulations are impossible because the computation costs are prohibitive. Several techniques have been developed to speed up simulations. Existing techniques are surveyed below.

Current techniques for accelerating discrete simulation programs, and their respective disadvantages, include:

- *Vectorization techniques* attempt to apply supercomputers to discrete event simulation problems. Although some success has been obtained in time-

driven simulations of homogeneous systems, vectorizing hardware provides little speedup for most *event-driven* simulations of practical interest [3].

- *Functional specialization* uses dedicated functional units to implement specific *sequential* simulation functions (e.g., event list manipulation) [7]. This method can provide only a very limited amount of speedup.

- *Replicated trials* execute independent, *sequential simulation programs* on different processors. This approach has two serious drawbacks: (1) it is only useful in Monte Carlo simulations [15] and studies testing multiple parameter settings, and (2) it is not scalable because the memory requirements increase in proportion with the number of parallel trials. In partitioned memory parallel processors, sufficient memory must be available in *each* processor to hold the entire simulation program.

- *Parallel time-driven simulations* execute events occuring at the same time step in parallel. This technique is very effective when the time step can be made sufficiently large that many events can be processed in parallel without sacrificing fidelity. Special purpose logic simulation engines have been very successful in exploiting this technique [28,11,23]. However, time-stepped simulation quickly degenerates to serial execution in asynchronous problem domains.

- *Parallel event-driven simulations* execute events in different processes in parallel. These events need not have the same event times. Two approaches, *conservative* and *optimistic*, have been adopted in these algorithms. *Conservative approaches* [22] can achieve speedup in certain situations [12]. However, existing approaches are limited to simulations with *static* processes and interconnection patterns. Further, empirical studies of deadlock avoidance and

deadlock detection and recovery algorithms indicate that performance is poor in many simulations of practical interest [12,25].

*Optimistic parallel simulation algorithms* such as Time Warp avoid many of the problems associated with conservative methods. Processes may advance relatively independently of one another, enhancing parallel execution [19]. Dynamic task creation and communication patterns are allowed. The central disadvantages of Time Warp are: (1) rollbacks may be frequent, and (2) the overheads associated with Time Warp, particularly for state saving, are substantial for simulations containing large amounts of state. This is problematic because simulation programs are notorious "memory hogs."

The thesis will focus only on parallel discrete event simulation. In the rest of this chapter, current parallel discrete event simulation algorithms are introduced, the advantages and disadvantages of each algorithm are discussed, and existing approaches to hardware support for simulation are investigated. Finally, a simulation engine is proposed for efficiently implementing the optimistic Time Warp mechanism.

## 1.3 Parallel Discrete Event Simulation

Recently, parallel simulation has become realizable by the emergence of powerful, highly parallel computers at relatively modest costs. In parallel simulation, each *physical process* of the system being simulated is modeled by a *logical process* in the simulator. Logical processes may be executed concurrently. The interactions between physical processes are simulated by messages passed between logical processes. Each message contains a timestamp indicating the point in simulated time at which the corresponding physical interaction occurs. *Timestamped messages* are identical to the events discussed earlier.

Unlike traditional sequential discrete event simulation programs, in which only a single global clock is used, each logical process in the parallel simulation has its own clock. Synchronization among these local clocks is a difficult problem. Several alternatives have been proposed, of which the Time Warp mechanism appears to be the most attractive. Time Warp offers the potential for more widespread exploitation of parallelism and relaxes many assumptions required by conservative approaches. Below, the clock synchronization problem is discussed in more detail and the existing solutions are surveyed.

### 1.3.1 The Clock Synchronization Problem

A critical problem that must be resolved by the distributed simulation mechanism is the management of simulated time. The simulator must ensure that it accurately models causality in the system being simulated. It is sufficient to guarantee that each process always processes incoming messages in non-decreasing timestamp order. This is a difficult task because, in general, each process is uncertain as to what messages will be sent to it in the future. In "conservative" simulation approaches, each process is forced to wait until it can determine with absolute certainty the next message that should be processed. This waiting, called *artificial blocking*, differs from the usual notion of blocking in parallel programs which results from waiting for data dependencies in the computation to be satisfied. All of the processes in the simulation program may become blocked (artificially or otherwise), causing the simulation program to become *deadlocked*. Deadlock is an important problem that must be addressed by conservative simulation strategies. Mechanisms to address this problem include deadlock avoidance and detection and recovery techniques.

### 1.3.2 Solutions to the Deadlock Problem

In conservative strategies, several assumptions concerning the parallel computer and the behavior of logical processes are required:

- Messages from one process to another arrive at the destination in the order in which they were sent.
- Messages are transmitted through the network without error, and no messages are lost.
- The sequence of timestamps on messages sent from one process to another forms a non-decreasing sequence of values.
- No logical processes are created dynamically.
- Communications among logical processes are statically defined.

Later, it shall be seen that these restrictions can be relaxed when using the Time Warp mechanism.

**1.3.2.1 Deadlock avoidance:** Chandy and Misra [5] developed a deadlock avoidance technique based on sending NULL messages. A NULL message is a "dummy" message (i.e. it does not correspond to any event in the simulation) that provides a lower bound on the timestamp of the next message sent from one process to another. To avoid deadlock, an additional notion called *lookahead* must be introduced. A process is said to have a lookahead of $L$ if an incoming message with timestamp $T$ cannot cause a new outgoing message to be generated that has a timestamp less than $T+L$. In general, the lookahead for a process varies dynamically during the simulation and depends on the type of incoming events.

In Chandy and Misra's algorithm, a process's lookahead is used to derive a lower bound on the timestamp of the next message sent by that process to its neighboring

processes. This mechanism provably avoids deadlock as long as no cycle exists with a lookahead of zero. Further details of the algorithm are discussed in [5]. Empirical evidence indicates, however, that this algorithm performs poorly if processes have poor lookahead ability [12].

**1.3.2.2 Deadlock detection and recovery:** Another strategy for attacking the deadlock problem uses a deadlock detection and recovery paradigm [6,9]. The simulator is allowed to proceed until it deadlocks. The deadlock is detected and broken, and the simulator is allowed to proceed again. A special process, called the controller, is used to detect deadlocks and invoke the recovery mechanism. Although this strategy does not require lookahead to ensure correct operation, empirical data suggests that like the deadlock avoidance method, it too relies on good lookahead ability to achieve good performance.

**1.3.2.3 Formalisms for parallel simulation:** Work by Chandy and Misra defines an elegant formalism for the distributed simulation problem that is based on *conditional-knowledge* [4]. Conditional-knowledge is information that will become true if some predicate is true, e.g., a submarine will fire a torpedo if it is not first destroyed by a missile. From this perspective, the attractiveness of Time Warp is based on its ability to process *conditional* knowledge; in contrast, conservative strategies must first convert conditional knowledge to unconditional knowledge before it can be processed. Simulation algorithms based on conditional knowledge are expected to enable speedup in certain (but not all) problem domains. Further, recent work by Aahlad and Browne casts the range of parallel simulation algorithms along a continuum [1]. This work may lead to useful analytic performance evaluation techniques and new parallel simulation mechanisms.

### 1.3.3 Time Warp

Both approaches (deadlock avoidance or detction and recovery) are conservative in that processes cautiously advance their clocks only when they can determine with absolute certainty that violation of causality constraints will not occur. The central drawbacks in these approaches are the artificial blocking phenomenon and the restrictions cited earlier.

The Time Warp (TW) [2,18,19] mechanism takes a more "optimistic" point of view. It allows processes to progress forward as rapidly as they wish, possibly risking violation of causality constraints. If an error does occur, i.e. messages are processed in the wrong timestamp order, the computation is *undone* by *rolling back* to a point in simulated time before the error. The simulator then proceeds forward until the next error occurs.

Time Warp offers the following advantages:

1. The TW mechanism allows greater exploition of parallelism. The artificial blocking phenomenon is eliminated.

2. The communication pattern between processes may be dynamic, in contrast to existing conservative simulation algorithms. Further, new logical processes can also be created dynamically.

Three data structures are employed by Time Warp to enable rollbacks to undo the computation. Each process must maintain (see figure 1.2):

- an input queue, which holds received messages that either have been processed or are waiting to be processed.

- an output queue, which list outgoing messages sent by the process.

- a state queue, which records previous states of the logical process that may be restored on rollback.

**1.3.3.1 Message processing and anti-messages:** The input message queue stores messages received by the process in timestamp order. A variable holds a pointer to the current message, i.e., the message now being processed, or the last message processed if the logical is not being executed, LVT (local virtual time) is defined to be the timestamp of the current message.

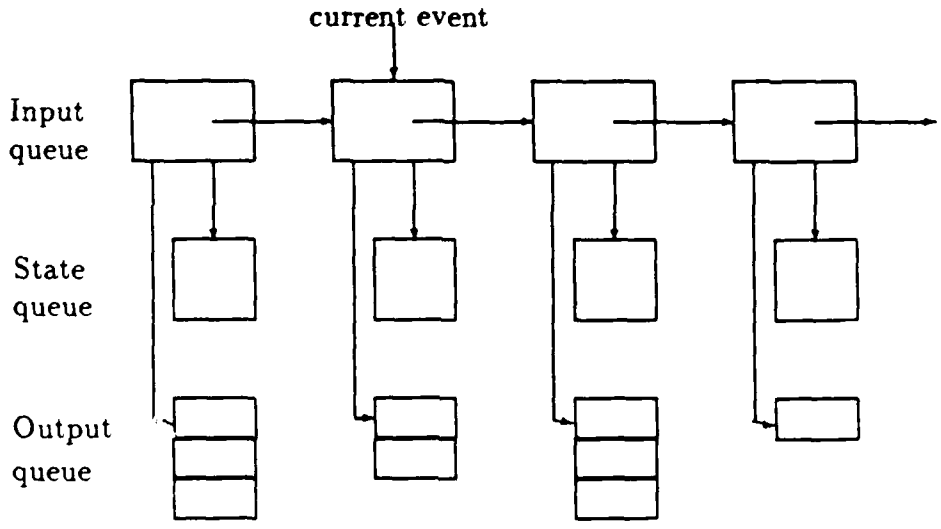



Figure 1.2. Three queues of a logical process

On each message arrival, the timestamp of the arriving message is compared with LVT. If it is greater than or equal to LVT, the arriving message is simply enqueued in the input message queue. Otherwise, the logical process must rollback, i.e., undo the effect of all messages in the input queue that lie between the new message and the current message.

Processing a message typically modifies the state of the logical process, and causes one or more new messages to be sent. The rollback necessitates restoring the logical process's state to a previously saved state in the state queue, and sending *anti-messages* for rolled back messages stored in the output message queue. An anti-message is logically identical to an original "positive" message except a flag indicates it is an anti-message. When a message is sent by a logical process, its corressponding anti-message is also enqueued in the output message queue.

When an anti-message is received, one of three possible cases will arise:

1. the corresponding positive message is found in the input message queue and has already been processed.

2. the positive message is found in the input message queue and has not yet been processed.

3. the positive message is not in the input message queue.

In case (1), the receiving logical process must rollback and annihilate the matching message, anti-message pair. In case (2), only annihilation is required. In case (3), the anti-message is enqueued into the input message queue till its corresponding positive message is received, at which time both are annihilated.

**1.3.3.2 Checkpoints and rollback:** In order to allow rollback, the state of each logical process must be periodically saved. Checkpoints are defined for this purpose, and indicate the time at which a process records its state. States at checkpoints must be kept in the state queue. Typically, a checkpoint will occur after processing each event.

**1.3.3.3 GVT and fossil collection:** The Time Warp uses a mechanism called *fossil collection* to reclaim memory resources that are no longer needed.

Fossil collection reclaims very "old" entries in the state queue and the input/output message queues. To perform this function, a bound must be derived on the "furthest" possible rollback.

This bound, called global virtual time (GVT), is defined to be the minimum timestamp of any unprocessed message or anti-message residing in system, either in queues or in transit. Memory used by messages, anti-messages and state queue entries with timestamp less than GVT can be reclaimed, that is, all except the state queue entry immediately prior to GVT.

Besides reclaiming memory resource, fossil collection also performs irreversible operations such as physical output (e.g. messages to output devices). Also, execution errors are only reported to the user when GVT exceeds the simulated time at which the error occurred.

Even though the TW approach offers great potential and flexibility, it may still fail to achieve good speedup because:

- Rollback may occur frequently.
- The state saving overhead necessary to allow rollback can be great.

The first problem, frequency of rollback is beyond the extent of this thesis. The second, state saving overhead, is required even though rollbacks occur infrequently and is the subject of this thesis.

State saving is very expensive in many applications. Assume that the data bus is 32-bits wide and the memory cycle time is 200 nanosecond. Copying only a modest amount of state, say 10k bytes, requires a minimum of one millisecond, assuming the memory is utilized 100% and no time is required for instruction fetches. Existing simulators require less than a millisecond to process each event in many applications[13], so this represents an overhead of more than 100%. Further, many simulations contain objects with over a megabyte of state, making copying infeasible.

#### 1.3.3.4 Other approaches to state saving (software and hardware):

Several possible state saving strategies have been explored. The most straight forward is to copy the entire state of the process after simulating each event. This approach is only practical if the state vector is small. Further, it is inefficient in memory utilization if only a small portion of the state is modified by each event.

A better approach is to use incremental copying, i.e. only the modified portion of state is copied. Determination of what portions of state may be modified can be done either during *compilation* or during *execution (runtime)*. The former approach uses data flow analysis to determine state variables that might be modified by an event, and embeds code into the program that only saves these variables before processing the event. Unfortunately, not every modified variable can be determined at compilation time. For example, if state variable $X$ is an array and $X[i]$ is known to be modified by an event, the index $i$ cannot be determined until runtime, so the entire array must be saved.

Alternatively, state variables might be saved at runtime just before they are modified. This will incur a significant runtime overhead, however. Also, rollback becomes very expensive, and may require an extensive amount of copying.

Another approach is to use infrequent (relative to state changes) copying, i.e. save the entire state of the process only after processing many events. The problem with this approach is that the rollback may have to be longer than is strictly necessary in order to reach the last saved state. If rollbacks tend to be short (e.g., only one or two events), this approach becomes exetremely inefficient. Empirical data suggests that in many practical applications, rollbacks are in fact usually only one or two events long [17,13].

Approaches using special purpose hardware have been proposed to alleviate the inefficiencies associated with software based approaches. Lee, Ghani and Heron built a recovery cache for the PDP-11 family of machines [20]. A recovery cache is

designed to be a special function unit that intercepts bus operations made by the PDP-11 CPU to its memory modules. When a recovery block has been entered and a variable is about to be modified (for the first time), the original value of that variable is copied to a special memory before it is actually updated. On rollback, the saved data in the recovery cache is restored. Although only incremental copying is required in [20], additional overhead (a memory read must precede each write) is introduced. This recovery cache, like that reported in [20], reduces the cost of state saving overhead at the expense of the rollback operation – extensive copying may be required on each rollback. While this is reasonable for fault-tolerant computation where errors are assumed to occur infrequently, it is not appropriate for many Time Warp programs.

Feridun, Lee and Shin used hardware recovery blocks in constructing a fault-tolerant multiprocessor [10,21]. Each node of this configuration has multiple state-save units, controlled by a monitor switch. Each state-save unit can hold a valid state. On detection of an error, the multiprocessor reconfigures itself, and the process assigned to the faulty module retreats to one of the previously saved state in a state-save unit. State saving in [10,21] is performed by excessive copying, however.

## 1.4 Summary of the State of the Art

Reviewing the state of the art for speeding up the simulation programs, it is clear that

1. *at present, no viable speedup techniques exist for many important simulation problems of vital interest.* Although the problem of speeding up discrete event simulation programs has been widely studied, all known parallel algorithms have serious deficiencies. None show good promise for large asynchronous simulations that contain objects with a large amount of state (the Achilles heel

of Time Warp) and either require dynamic objects/interconnection patterns or contain unpredictable interdependencies (where conservative strategies fail). Many important problems, e.g., many large battle simulations with embedded continuous simulation models, contain precisely these characteristics.

2. Among the available techniques, Time Warp offers the greatest flexibility and potential for speed up. The reduction of rollback in Time Warp is fundamentally a scheduling problem, and is not addresses here. This thesis addresses the state saving problem by using sophisticated memory management hardware.

In this thesis, the *rollback chip*[1] (RBC) is proposed to attack the state saving problem. Ideally, the RBC would be used in a special simulation engine, such as Utah Simulation Engine (USE) described later. However, it can also be added to existing multiple processors machines (e.g., the Intel iPSC) as a special memory board that provides hardware assistance for state saving and rollback.

The value of the rollback chip relative to a software based implementation of state saving and rollback using copying is greatest when (1) the amount of state is large (e.g., a megabyte), and/or (2) the application makes checkpoints very frequently (e.g., every few hundred microseconds). However, as noted earlier, copying may represent a significant overhead even for modest sized state.

## 1.5 Simulation Hardware

Several special purpose simulation machines have been constructed. An increase of 10 to 1000 times in speed can be gained by applying these machines to logic simulation. Three special purpose simulation machines will be briefly described next. Unlike the simulation hardware described in this thesis, these machines are

[1] The name "rollback chip" is actually somewhat of a misnomer because current circuit densities preclude a *single chip implementation*. Nevertheless, this terminology is used because a single chip implementation is expected to be feasible in a few years.

based on time-driven mechanisms, and thus are not appropriate for asynchronous applications.

### 1.5.1 Existing machines

**1.5.1.1 The Yorktown Simulation Engine (YSE)** The Yorktown Simulation Engine is a logic simulation machine developed by IBM [8,11,23,24]. The YSE can have up to 256 processors, plus an array simulator used for emulating main memory, cache, or register files. The processors communicate with each other during the simulation over a high speed crossbar network. The machine has a toal gate capacity of 2 million gates and can evaluate approximately 3 billion gates per second. However, the YSE supports only unit delay simulation.

**1.5.1.2 The Hardware Logic Simulator (HAL)** The Hardware Logic simulator was designed by Nippon Electric Corporation (NEC) [11,27]. The HAL is similar to the YSE configuration in that it only supports unit delay simulation. It uses a global clock and multiple processors. It also has fewer processors (29 plus 2 special processors for memory simulation). Unlike YSE, HAL only recomputes the outputs of those gates that have input signal changes during the current time step. The total capacity of HAL is nearly 300,000 gates, and its speed is about 1.2 billion gate evaluation per second.

**1.5.1.3 The Zycad Logic Evaluator** The Zycad Corporation's Logic Evaluator [28,11] has a maximum of 16 processors which communicate with each other via a single high speed bus. Communication to the host computer is via a slower bus. The system can model three logical levels and three logical strengths. Thus, both gate level and switch level simulations can be performed. It has a gate capacity of 1 million gates, and a speed of 15 million events per second.

### 1.5.2 The Utah Simulation Engine (USE)

Unlike existing simulation engines, the Utah Simulation Engine is intended to enable solution of large-scale *asynchronous* simulation programs.

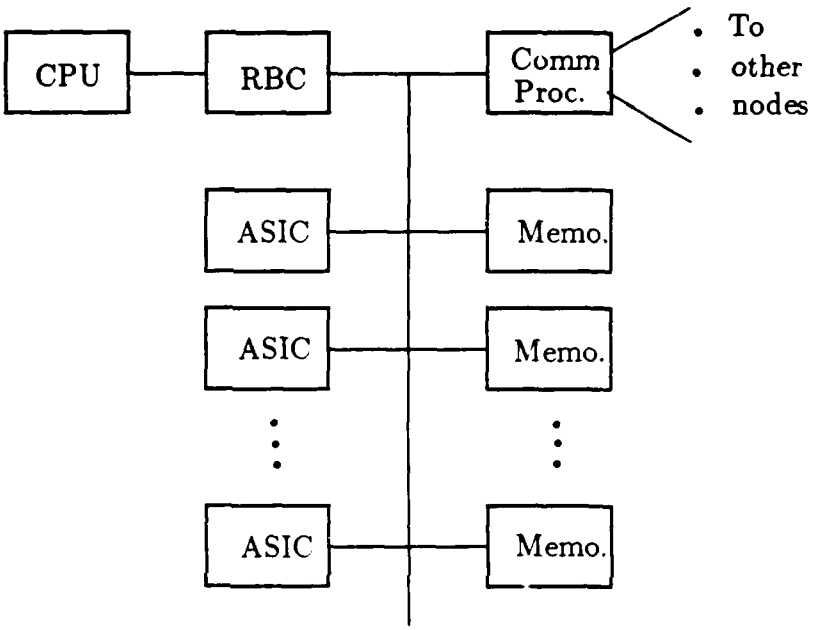



Figure 1.3. One node of Utah Simulation Engine

The USE is based on the Time Warp mechanism. It is an MIMD parallel computer. Each node of this message-based multicomputer architecture consists of a general purpose CPU, a communication processor, several memory modules, one or more application specific integrated circuits (ASIC) and the rollback chip (see figure 1.3.). Intended as a backend processor, this USE is programmed using familiar *object-oriented* or *process-oriented* simulation methodologies.

## 1.6 Organization of the Thesis

This thesis will describe the rollback chip in detail, including:

- Chapter 2 presents the functional description of the rollback chip.
- Chapter 3 describes the mechanisms used by the RBC to efficiently implement the functions defined in Chapter 2.
- Chapter 4 describes one design of the RBC.
- Chapter 5 discusses results of extensive performance evaluation studies of the RBC.
- Chaper 6 summarizes the results of this thesis.

# CHAPTER 2

# THE ROLLBACK CHIP

## 2.1 Interface to the Rollback Chip

The rollback chip (RBC) implements the state saving and rollback operations for a single node of a multi-processor computer. It provides each simulation process with a data segment called *version controlled memory*, or *VCM*. The simulation process stores all of its state variables in this data segment. Version controlled memory has identical semantics as ordinary read/write memory, except that the process may, at any time, "mark" the state of the memory as one that it may later want to restore via a ROLLBACK operation. A state marked by the process is known as a *version* of that state. Typically, the process will issue a MARK operation after it finishes processing each event (message).

The RBC only acts on memory references to version controlled memory. VCM is assumed to have a fixed maximum size, and in the discussion that follows, is assumed to be statically mapped into the processor's address space. Specific numbers for the various characteristics of VCM are used to make the discussion more concrete, and to indicate typical values. In particular, each VCM data segment is assumed to contain up to 4M bytes of storage, and up to 64 VCMs are allowed in each *processing element (PE)*.

The RBC supports six operations: RESET, memory READ, memory WRITE, MARK, ROLLBACK and ADVANCE. These six operations, along with their arguments, are generated by the node processor, and are passed to the RBC for

execution. RESET, MARK, ROLLBACK and ADVANCE are generated by writing into the RBC's control registers (which in turn may be mapped into the processor's address space), and READ and WRITE are CPU accesses to variables that have been mapped by the compiler into VCM. Each operation is described below:

**RESET.** Initialize the rollback chip prior to the execution of a Time Warp program.

**MARK.** Mark the current state of version controlled memory.

**WRITE(A,D).** Write data D into memory address A.

**READ(A):D.** Read the most recently written version of data associated with address A (excluding rolled back write operations) and return this data D to the CPU.

**ROLLBACK(k).** Restore the version controlled memory to the k-th previously marked state (k>0).

**ADVANCE(k).** The k oldest marked states are no longer required, and can be fossil collected. During fossil collection, resources that are no longer needed are reclaimed, and irrevocable operations, such as I/O are performed.

## 2.2 The Memory Address Space

The RBC partitions the address space seen by the CPU into two types of memory: VCM and non-VCM (see figure 2.1). The RBC only manipulates references to variables stored in the VCM area. The VCM area is further divided into several contiguous data segments. As mentioned earlier, each VCM data segment is 4M bytes in length, and a data segment is assigned to each process which is executed on the CPU. Addresses generated by the CPU implicitly indicate whether it refers

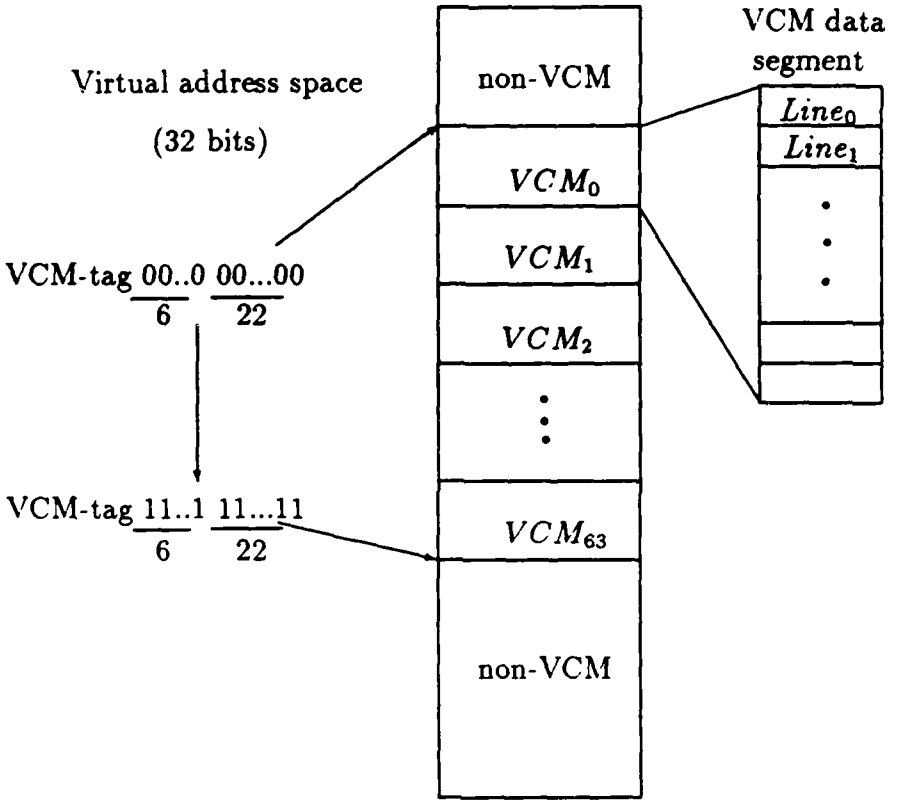
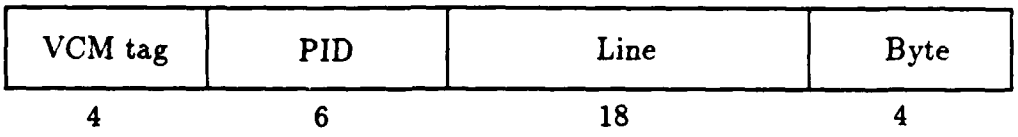



Figure 2.1. Virtual memory space seen by the RBC

to VCM or not, and if it does refer to a VCM, the RBC is activated to process the memory request to which data segment.

An address format that contains 32 bits is assumed in the following discussion, as shown in figure 2.2. As the address format shows, the RBC can support 64 processes (6 bits), 256K lines (18 bits) per process, and 16 bytes (4 bits) per line. The VCM_tag field is used to distinguish the VCM references from the non-VCM ones.




* VCM tag: flag to indicate references to VCM
* PID: process identification number
* Line: line number
* Byte: byte in line

Figure 2.2. Address format used by the RBC

## 2.3 RBC Data Structures

### 2.3.1 Mark Frames

The rollback chip must maintain different versions of each state variable to enable a previous version to be restored. Each version of a state variable is stored in a separate area of (virtual) memory called the *mark frame*. Each mark frame has the same size as a version controlled memory data segment, and is divided into some number of fixed length *lines*. An RBC line is similar to a line in a cache memory system; it is transparent to the processor and serves as the quantum of data accessed on each memory reference. Here, lines are assumed to be 16 bytes in length.

Mark frames are organized as a circular list (see figure 2.3). New frames are added to the top of the stack, while outdated frames can be discarded from the bottom of the stack. The circular list implementation simplifies storage reclamation and reuse. Overflows of the circular list will be discussed later, but will be ignored for now. The *CMF (current mark frame)* refers to the top frame of the stack. Similarly, the *OMF (oldest mark frame)* refers to the bottom frame of the stack.

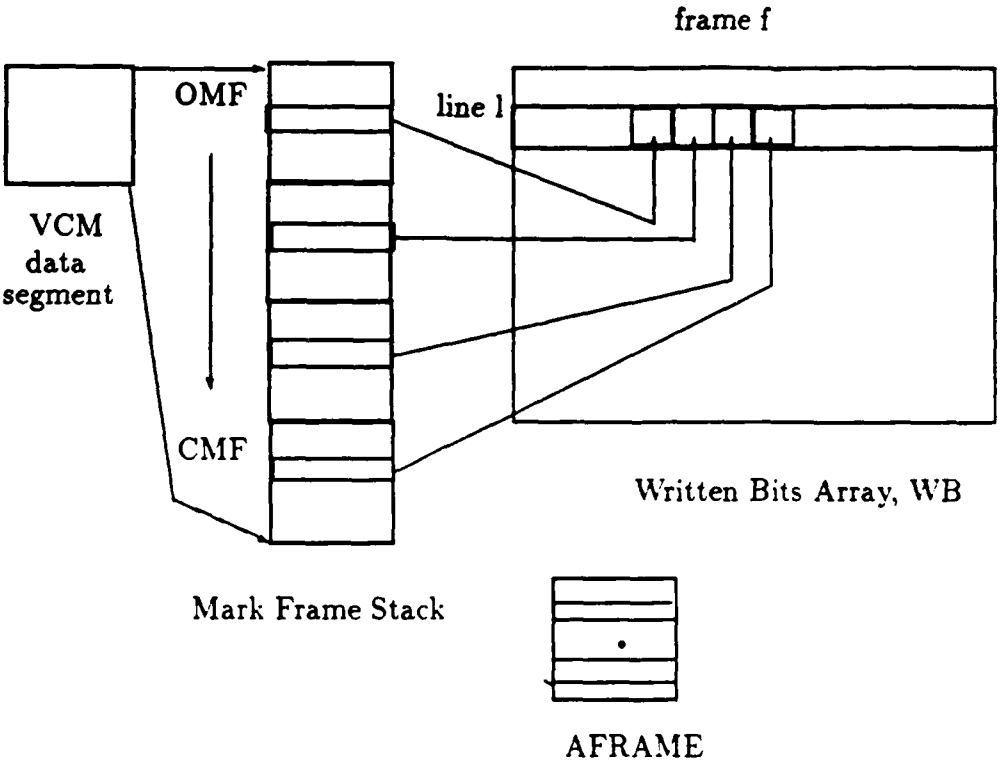



Figure 2.3. Data structures used by the RBC

The RBC contains the OMF and CMF registers which contain pointers to these to frames respectively.

In the discussion that follows, all arithmetic is assumed to be modulo the size of the mark frame stack. Explicit modulo operators have been deleted to simplify the presentation, but are included in the rollback chip simulator that is included as an appendix. Finally, the discussion that follows pertains to a single VCM. Extensions to handle multiple VCMs per processor are straightforward, and be will be discussed later.

The six operations defined earlier in this chapter can be easily explained in terms of this stack-based implementation. The RESET operation resets the CMF and OMF registers to 0. The MARK operation pushes a new frame onto the

stack by incrementing the CMF register. *No* data is copied on MARK operations. Memory WRITE operations write the data into the current mark frame, while READ operations must scan through the stack starting from CMF to find the most recent version of the variable. Actually, as will be discussed in detail later, the RBC caches recently used *most recent version (MRV)* data to reduce the amount of searching that is required in practice. Finally, ROLLBACK($k$) pops $k$ frames from the stack by decrementing the CMF register by $k$, and ADVANCE($k$) removes the $k$ oldest frames by advancing the OMF register.

Two additional aspects of the RBC must be discussed. First, since not every variable in the frame is written between successive MARK operations, mark frames will usually contains "holes" where no valid data exists. A flag is required to indicated which lines contain valid data, and which lines do not. Secondly, the ADVANCE operation may accidentally discarded needed data, so additional precautions must be taken. These two aspects are described next.

### 2.3.2 Written Bits

A single bit called a *written bit* is associated with each line of each frame in the stack. The written bit is used as a flag to indicate whether or not the line contains valid data. It is set (i.e. 1) when the line contains valid data. To improve the efficiency of searches for the most recent version of a line, the written bits are organized as a two dimensional array. Each array entry WB[$l$,$f$] holds the written bit corresponding to line $l$ of mark frame $f$ (see figure 2.3). The search for the most recent version of line $l$ is accomplished by searching row $l$ of the written bit array starting at WB[$l$, *CMF*] until a set bit is found (see figure 2.4).

### 2.3.3 The Seldom Written Data Problem

As mentioned earlier, simply incrementing the OMF register on each ADVANCE operation may cause needed data to be discarded. To see this, consider the following

```
Search_MRV (line ln) ::
     for i:= CMF to OMF by -1 do
        if (WB[ln, i] == 1) Return(i);
     end-for;
     Return (ERROR);     /* all zero written bit is error state */
end Search_MRV;
```

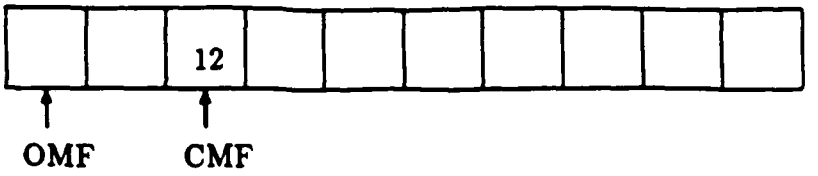
Figure 2.4. Program to locate the most recent version of line. In practice, a block of (say) 16 written bits are read in parallel and scanned using a priority encoder.

situation: a variable is written infrequently, so its most recent version of data becomes buried far into the mark frame stack. If the OMF register advances beyond the frame containing this data and discards that frame, this data will be erroneously discarded, and the most recent version of that variable is lost. This is illustrated by the example in figure 2.5.

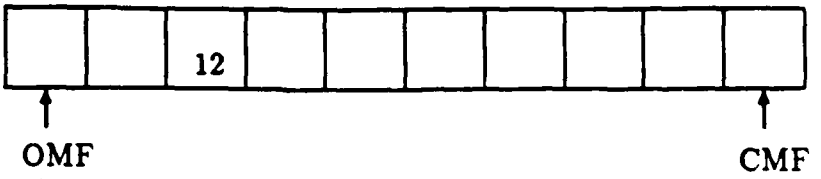
To avoid this problem, a special mark frame called the *archive frame* is introduced. The most recent version of line $l$ that is older than OMF is copied to the archive frame before the frame is discarded by the ADVANCE operation. If a READ operation fails to find a set written bit, it assumes the most recent version of the variable is stored in the archive frame.

## 2.4 RBC Operations

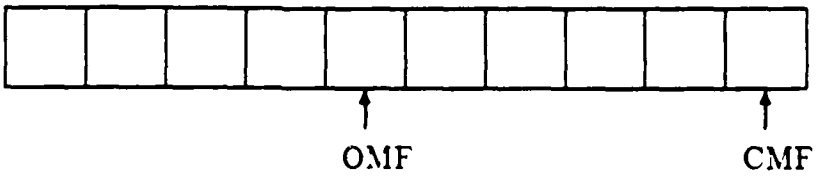
The algorithm implemented by the RBC is shown in figure 2.6 using a Pascal-like syntax. MRV denotes the frame number holding the most recent version of the line in question, and may refer to the archive frame. The description is straightforward. One point worth noting is that the WRITE operation must first copy the MRV line into the CMF if the CMF written bit is not set. This is required because WRITE operations do not overwrite the entire line.




(a) A value is written in CMF.




(b) After several mark operations, the value is deeply embedded in the stack.




(c) Advance operations erase needed data for the seldom written variable.

Figure 2.5. Example: seldom written data lost on ADVANCE operation

Some operations may initially seem to be very slow, and/or inefficient. Special mechanisms have been developed to efficiently implement the RBC algorithm shown in figure 2.6. These mechanisms are described in next chapter.

```
/*****************************************************************/
/* N_Lines is the number of lines in VCM, and N_Frames is        */
/*      the number of frames in the mark frame stack.            */
/* Line is a data type corresponding to a line of data.          */
/* Memory references are word-based, so a Read request to        */
/*      address A (denoted as ln.A) will return a word of data*/
/*      from its MRV frame (denoted as Stack[ln][MRV].Word).  */
/*****************************************************************/

Boolean WB[N_Lines][N_Frames];
Line Stack[N_Lines][N_Frames];

Reset () ::
     CMF := 0; OMF := 0;
     for each line ln of each frame fr do
        WB[ln][fr] := 0;
     end-for;
end Reset;

Write (address ln.A, data D) ::
     if (WB[ln][CMF] == 0) /* first write to the line of CMF frame */
        Stack[ln][CMF] := Stack[ln][MRV];
     Stack[ln][CMF].Word := D;
     WB[ln][CMF] := 1;
end Write;

Read (address ln.A) ::
     Return (Stack[ln][MRV].Word);
end Read;

Mark () ::
     CMF := CMF+1;
end Mark;

Rollback (k frames) ::
     for each line ln do
        for each frame fr := CMF-k to CMF+1 do
             WB[ln][fr] := 0;
        end-for;
     end-for;
     CMF := CMF-k;
end Rollback;

Advance (k frames) ::
     for each line ln do
            /* OMRV is the MRV frame older than OMF+k */
         if (OMRV frame exists)
             AFrame[ln] := Stack[ln][OMRV];
     endfor
     OMF := OMF+k;
end Advance;
```

Figure 2.6. A cache-based rollback chip operations

# CHAPTER 3

# MECHANISMS FOR EFFICIENT IMPLEMENTATION

Implementation is a challenging problem because several aspects of the algorithm in figure 2.6 will be unacceptable inefficient, slow and/or inflexible if implemented in the obvious way. Potential difficulties with the algorithm described in the previous chapter (and proposed solutions) are:

**Overflow of the mark frame stack.** The number of versions of each variable that may be required during a simulation is unbounded. Further, it is more efficient for the RBC to manipulate a "block" of versions at one time (particularly when searching through the written bits) rather than individual versions. To address these issues, *working areas* are introduced to partition the mark frame stack. These working areas, as well as the entire mark frame stack. are organized as a circular list.

**Slow access to MRV data.** Accesses to MRV data require a search through the written bit matrix. To address this problem, the most recent version of recently used lines are cached in the rollback chip, allowing READ and WRITE "hits" to be performed at conventional cache memory speeds. An additional optimization is introduced to reduce the search time required for a cache "miss".

**Slow ROLLBACK operation.** Many written bits must be reset on each rollback. An efficient mechanism called rollback histories has been used to avoid

updating the written bits that do not reside in the cache when a rollback occurs; the bits are instead cleared when they are reloaded into the cache.

**Slow ADVANCE operation.** Extensive copying may be required for the ADVANCE operation to move data into the archive frame. To address this problem, the rollback chip processes ADVANCE operations in parallel with other RBC activities. The processor need not wait for the ADVANCE operation to complete unless it runs out of memory. Also, an optimization is introduced to reduce the amount of data that must be copied to the archive frame.

**Poor memory utilization.** If few state variables are modified between MARK operations, most of the memory in the mark frame stack is wasted. A dynamic memory allocation scheme based on demand paging is used to allocate physical memory only when it is needed.

**Slow context switches.** Mechanisms similar to those used in translation lookaside buffers in memory management units (MMU) avoid excessive overheads if there are many simulation processes mapped to a single processor.

The current RBC design has mechanisms to address these issues. Before discussing these aspects in detail, earlier RBC designs are briefly presented.

## 3.1 Earlier RBC Designs

During the development of the rollback chip, several different designs have been developed and evaluated. The final design is based on a special type of data cache, and offers significant advantages over previous ones. Nevertheless, these earlier designs will be described next in order to document alternative approaches that were considered.

Originally, the RBC was envisioned as a single chip containing the entire written bit array. The second design of the RBC is analogous to an MMU (memory management unit). In this latter version, the written bits were stored in conventional RAM and only recently referenced bits were cached in a TLB (table lookaside buffer). Like the initial design, the RBC simply translated virtual address (addresses in VCM) generated by the CPU to physical address (addresses in the mark frame stack). This translation essentially involved concatenating a frame number to the address generated by the CPU. The values of state variables are then directly read (written) from (to) memory. In the current design, the RBC functions as a data cache, i.e., in addition to WB information, the most recent version of each recently used state variable is also cached in high speed memory.

### 3.1.1 A WB-based Rollback Chip

The focus of this initial design was on avoiding excessive copying for state saving, and rapid searches for the most recent version of the referenced variables. Therefore, the entire written bit array was maintained in the RBC.

A prototype is described in [14]. A 16X16 WB array is used to perform the state-saving for a single process with VCM size of 16 lines (and at most 16 mark frames are allowed at any time).

Because most Time Warp programs need larger VCMs and more than 16 versions of state, extensions were developed to handle programs with a larger VCM size and more versions than that which could be fitted on the chip (see [14]).

The central disadvantage of this design is that it does not support multiple processes per processor, and the proposed extensions increase the complexity of the hardware considerably.

### 3.1.2 An MMU-based Rollback Chip

Because of the deficiencies in the first design, a refined design was proposed. A TLB was used to cache the address translation information necessary to map references to VCM to the appropriate mark frame. The superiority of this refined design is that it can easily accommodate programs with larger state spaces, and multiple processes per processor.

The written bits are still used to indicate valid data. However, only individual rows of bits are stored with other address translation information in the entries of the TLB. The details of this design can be found in an appendix.

Memory READ and WRITE operations in this design always access main memory. The central drawback of this approach is that all memory accesses to VCM are slowed because they must first pass through the RBC.

The current design uses a data cache to address the memory latency problem. The remainder of this thesis will deal exclusively with this cache-based design.

## 3.2 Working Areas and Dynamic Growth of the Mark Frame Stack

The mark frames in the stack are partitioned into fixed sized working areas (WA). A working area contains (say) 16 contiguous mark frames. For example, the first working area consists of mark frame 0 through 15, the second 16 through 31, etc. Here, we assume the RBC supports 16 working areas or up to 256 versions of the process state. The working area number can be obtained by extracting the high order bits of the frame number; the lower order bits provide an offset within the working area. Assuming the frame number is 8 bits, the upper nibble indicates the working area, and the lower nibble the frame within the working area. The *oldest working area (OWA)* is defined to be the working area containing the OMF.

Similarly, the *current working area (CWA)* is defined as the working area containing the CMF.

Using working areas, it is possible to devise a scheme to allow the mark frame stack to dynamically expand beyond the fixed size allocated to the circular buffer. A set of *working area registers* can be defined in the rollback chip, each of which points to a single working area of the mark frame stack. Like the mark frame stack, the working area registers are organized as a circular queue. When the stack overflows, registers corresponding to working areas at the bottom of the stack are saved in memory, allowing these registers to be used to accommodate the expanding stack. These saved registers would be eventually garbage collected by ADVANCE operations. Alternatively, a very long rollback might cause this saved information to be reloaded back into the working area registers.

Though feasible, supporting dynamically expanding stacks adds a nontrivial amount of complexity to the rollback chip design. Also, context switches become more expensive, necessitating saving and restoring the contents of the working area registers when execution switches to another process, or the use of multiple banks of registers. Further, even if dynamic stacks are not supported, overflow of the mark frame stack can be easily handled by blocking the offending process until global virtual time advances sufficiently to allow old frames to be garbage collected and reused. It is unlikely that such blocking will diminish performance because processes running out of stack space are far ahead of others, so they are not likely to be on the critical path of the computation. In the discussion that follows, it assumes that the rollback chip does *not* use working area registers and supports a fixed sized mark frame stack.

## 3.3 The RB Cache

The READ operation must return the most recent version of the referenced data. Searching through the written bit array to locate the MRV frame on each READ operation is too expensive. A cache is used to address this problem.

Two approaches, one using a write-through policy and the other a copy-back approach were considered and compared. In the write-through approach, memory writes are forwarded to main memory immediately; the copy-back policy employs extra bits call *dirty bits* to record the change only in the cache, and updates the memory afterwards. The copy-back approach has the advantage that it generates less memory traffic. However, the write-through cache has a simpler circuit design. Although this latter point is true of conventional caches, the increase of complexity using copy-back is even greater in the RBC for reasons that will become clear later.

### 3.3.1 Design I: The Write-through Cache

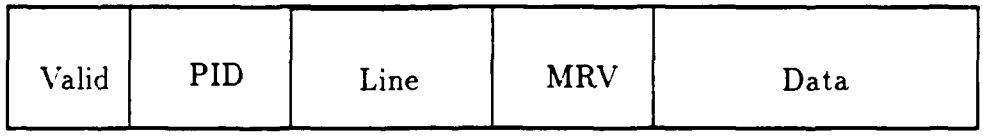



Figure 3.1. The fields in a write-through cache entry

The fields in each RB cache entry using a write-through policy are (see figure 3.1):

**Valid:** A single bit to indicate if the entry holds valid information.

**PID:** The ID of the process to which the line belongs. Each process is assumed to have a single VCM, so this field is actually the number of the corresponding VCM data segment.

**Line:** A field to indicate the line number cached in this entry. Associative searches are performed on this and the PID field.

**Data:** A field holding the most recent version data for the line.

**MRV:** The number of the frame that contains the most recent version of the line. This field is used in write and rollback operations.

The six operations defined earlier affect the cache as described below:

RESET: The Valid field of each entry of the RB cache is reset to 0, indicating that no valid information is held.

READ: The RB cache operation is similar to that of a conventional cache for READ operations. If a hit occurs, i.e., a valid entry is found in which the line number and PID fields match that of the referenced address, the requested data is extracted from the Data field, and returned to the requesting process (see figure 3.2). This is identical to the operation of a conventional cache, so an access time equivalent to that in conventional cache memories can be expected. If a miss occurs, the least recently used line or an invalid entry (if there is one), is selected and overwritten. The frame containing the MRV of the line must be determined by searching the corresponding written bit array. The information for the referenced line is loaded into the cache, and the requested data is returned to the CPU. An optimization will be described later to reduce the MRV search time for RB cache misses.

WRITE: On write hits, the data are written into the cache as is done in conventional caches, and the CMF register is written into the MRV field. Because a

```
Read_Hit (cache entry e) ::
        Return (cache[e].Data);
end Read_hit;

Read_Miss (line ln) ::
     Find the LRU entry (or an invalid entry if there is one) e;
     Search the MRV frame for line ln;
     cache[e].Valid := TRUE;
     cache[e].Line := ln;
     cache[e].Data := Stack[ln][MRV];
     cache[e].MRV := MRV;
     cache[e].PID := PID;
     Return ([cache[e].Data);
end Read_Miss;

Write_Hit (cache entry e, line ln, data D) ::
     cache[e].Data := D;
     cache[e].MRV := CMF;
     Stack[ln][CMF] := cache[e].Data;
     WB[ln][CMF] := 1;
end Write_Hit;

Write_Miss (line ln, data D) ::
     Find the LRU entry (or an invalid entry if there is one) e;
     Search the MRV frame for line ln;
     cache[e].Valid := TRUE;
     cache[e].Line := ln;
     cache[e].Data := Stack[ln][MRV];
     cache[e].MRV := CMF;
     cache[e].Data := D;  /* only modify referenced word */
     cache[e].PID := PID;
     WB[ln][CMF] := 1;
     Stack[ln][CMF] := cache[e].Data;
end Write_Miss;

Rollback (destination frame dst) ::
     for each cache entry e do
         if (cache[e].PID==PID and cache[e].MRV > dst)
                cache[e].Valid := FALSE;
         end-if
     end-for
end Rollback;
```

Figure 3.2. Write-through cache operations

write-through policy is used, the line is also written immediately to the CMF frame of memory.

A write miss operation is essentially a read miss immediately followed by a write hit. The most recent version of the line must be loaded into the cache because write operations do not modified the entire line. The new data and the CMF register are then written into the cache, and the line is written into the CMF frame in memory.

Because read and write misses must search through the line's written bits to find the most recent version of the data, the miss penalty of the RBC could be significantly greater than that of conventional caches, especially when the most recent version of data was created long ago. A simple optimization can be used to speed up the search. Whenever an entry is deleted from the cache, the MRV field (actually just the working area that contains the MRV frame is necessary) can be written to memory in a special location associated with that line. An integer vector called *LastWA[]* is defined for this purpose; LastWA[$i$] is associated with line $i$. The search for the MRV frame can now begin at either CWA or LastWA, whichever is smaller (using modulo arithmetic). The LastWA may be greater than CWA because ROLLBACK operations may invalidate the LastWA information, i.e., LastWA does not necessary point to the working area containing the MRV frame. However, one would expect that LastWA will reduce the time required for MRV searches in most situations. One disadvantage of this approach is that an additional memory access is required on misses to read and write the LastWA information.

ROLLBACK: ROLLBACK operations must invalidate each cache entry whose MRV field is greater than the CMF (after the rollback). This can be implemented with a custom circuit with embedded comparison logic so that the cache entries can be updated in parallel.

### 3.3.2 Design II: The Copy-back Cache

The second design uses a copy-back policy to reduce memory traffic. This design also stores WBs in the cache to maximize the hit rate by reducing the number of invalidated entries on rollback. However, extra control circuits are required to achieve these features.

In addition to the fields listed in the first design (Valid, Line, Data, MRV and PID), the following are added:

**WB:** A vector of 16 written bits which correspond to the working area containing the MRV frame of the cached line.

**DirtyWB:** A flag to indicate if the cached written bits are the same as those stored in memory.

**DirtyData:** A flag to indicate if the cached data are the same as those stored in memory.

The rollback chip operations are implemented as follows (see figure 3.3):

READ: READ hits are handled in exactly the same way as the previous design. However, for READ misses, the written bits vector and/or data of the deleted entry must be written into memory if the corresponding dirty bit(s) are set. The MRV of the requested data is then located, and loaded into the deleted cache entry, and the requested data are returned to the CPU.

WRITE: A write hit is slightly more complicated than that of a write-through cache because the WB vector must be updated, and both the DirtyWB and Dirty-Data bits must also be revised. If the CMF and MRV frames are the same, the data can be written into cache as is done in the write-through cache, and the DirtyData bit is set. However, if the MRV frame is different from the CMF, the MRV data must be preserved for future rollbacks; that is, the data must be written to memory

```
Read_Hit (cache entry e) ::
        Return (cache[e].Data);
end Read_Hit;

Read_Miss (line ln) ::
     Find the LRU entry (or an invalid entry if there is one) e;
     if (cache[e].DirtyWB) WB[cache[e].Line][wa] := cache[e].WB;
     if (cache[e].Data)
            Stack[cache[e].Line][cache[e].MRV] := cache[e].Data;
     Search the MRV frame for line ln;
     cache[e].Valid := TRUE;
     cache[e].Line := ln;
     cache[e].Data := Stack[ln][MRV];
     cache[e].MRV := MRV;
     cache[e].PID := PID;
     cache[e].WB := WB[ln][wa];
     cache[e].DirtyWB := FALSE;
     cache[e].DirtyData := FALSE;
     Return ([cache[e].Data);
end Read_Miss;

Write_Hit (cache entry e, line ln, data D) ::
     if (CMF==MRV) then
        cache[e].Data := D;
     else
        if (cache[e].DirtyWB) WB[ln][wa] := cache[e].WB;
        if (cache[e].DirtyData) Stack[ln][MRV] := cache[e].Data;
        cache[e].Data := D;
        cache[e].MRV := CMF;
        cache[e].DirtyWB := TRUE;
        cache[e].DirtyData := TRUE;
        cache[e].WB[CMF] := 1;
end Write_Hit;

Write_Miss (line ln, data D) ::
     Find the LRU entry (or an invalid entry if there is one) e;
     if (cache[e].DirtyWB) WB[cache[e].Line][wa] := cache[e].WB;
     if (cache[e].Data)
            Stack[cache[e].Line][cache[e].MRV] := cache[e].Data;
     Search the MRV frame for line ln;
     cache[e].Valid := TRUE;
     cache[e].Line := ln;
     cache[e].Data := Stack[ln][MRV];
     cache[e].MRV := CMF;
     cache[e].Data := D;
     cache[e].PID := PID;
     cache[e].DirtyWB := TRUE;
     cache[e].DirtyData := TRUE;
     cache[e].WB[CMF] := 1;
end Write_Miss;
```

Figure 3.3. Copy-back cache on READ and WRITE operations

if the dirty data bit is set. In addition, the WB vector and MRV information must also be modified to reflect the new MRV frame,i.e., the CMF. If the CMF and previous MRV belong to the same working area, only the written bit vector is changed (and, of course, the DirtyWB bit is set). If they do not match, the written bit vector must be stored into memory (if the DirtyWB is set), and all written bits must be cleared except CMF, which now holds the most recent version of the data. These modification to the cache entry can be done in parallel. Finally, the cached data is modified with new data, and the DirtyWB and DirtyData bits are set.

The case when the CMF is not equal to the MRV frame in a WRITE operation only happens on the first write to that line after a MARK operation. Subsequent writes to the line follow the simplest case described above. Further, memory writes can be queued by the RBC, so the processor need not wait for memory write operations to complete.

As before, a write miss operation can be viewed as a read miss immediately followed by a write hit. The most recent version of the line must be loaded into the cache because write operations do not necessary modified the entire line.

ROLLBACK: Since the copy-back RB cache contains the WB vector, rollbacks need not invalidate all cache entries whose MRV frame is newer than the destination of the rollback (i.e. the new CMF). However, clearance of written bits and comparison are required and in some cases the MRV data must be reloaded back into the cache.

When a rollback occurs, the RBC first clears the written bits of the rolled back frames in each cache entry. This operation can be done in parallel using circuitry embedded in the cache for this purpose. After this is completed, each entry falls into one of the following cases (see figure 3.4):

1. No written bit is cleared, i.e. there were no set written bits among rolled back frames. Entries of this type require no further modification.

| Valid | Line | MRV | WB | Data |
|---|---|---|---|---|
| 1 | 38 | 36 | 0011100000000000 | 2780 |
| 1 | 47 | 43 | 0100100010010000 | 591 |
| 1 | 15 | 39 | 0000001100000000 | 3094 |

(a) Three entries before rollback. Other fields are not effected.

| Valid | Line | MRV | WB | Data |
|---|---|---|---|---|
| 1 | 38 | 36 | 0011100000000000 | 2780 |
| 1 | 47 | 36 | 0100100000000000 | 1200 |
| 0 | 15 | 39 | 0000000000000000 | 3094 |

(b) After rollback to frame 37.

(1) does not change.

(2) MRV is changed, reload new data (set DirtyWB).

(3) entry is invalidated.

Figure 3.4. Invalidation of cache entries on rollback.

2. One or more bits are cleared, but the cached bit vector still has one or more bits that remain set. These entries remain valid, however, new most recent version of data must be reloaded into the cache. Entries such as these are invalidated in the previous design, so a higher hit rate can be expected.

3. One or more bits are cleared, and the entire cached bit vector becomes cleared, i.e., no set bits remain. Entries of this case are invalidated by resetting the Valid bit to 0.

ADVANCE: ADVANCE operations do not change the cache, however, dirty entries (entries that have either DirtyWB or DirtyData is set, or both) corresponding to frames that are about to be fossil collected must be written into memory. This

action, called *flushing*, must be performed before the fossil collection of the working area begins, the fossil collection process then uses only the information stored in memory.

## 3.4 Rollback Histories

The RBC algorithm requires that a rollback operation clear all of the written bits corresponding to frames that are rolled back, i.e., popped from the stack. A brute force implementation of this operation will be too expensive for programs containing large amounts of state. An obvious alternative is to clear all written bits for new frames that are pushed onto the stack, however, this simply transfers the problem to the MARK operation, making it too expensive. The rollback history (RBH) mechanism is designed to efficiently clear the appropriate written bits when a ROLLBACK occurs.

The key idea used by the rollback history mechanism is that *no written bits stored in memory are cleared when a rollback occurs; instead, the written bits are cleared on the fly as they are read from the written bit memory (e.g., following a cache miss).* This dramatically improves the efficiency of the rollback operation, at the cost of a small increase in the cache miss penalty.

Using this "lazy" approach, the written bits in memory may not be updated until long after the rollback occurred. Therefore, the question that must be answered is "which written bits must be cleared when they are read from the written bit memory?" The *rollback history (RBH)* mechanism provides this information.

One can rephrase to above query to ask an equivalent question: "what is the deepest rollback that has occurred since the written bits were written into memory?" If the written bits were written to memory at time $t$ (meaning they were correct and up to date at time $t$), and the deepest rollback that has occurred since

time $t$ was to frame $f$, then the written bits that are newer than frame $f$ must be cleared. Therefore, one approach to this problem is to:

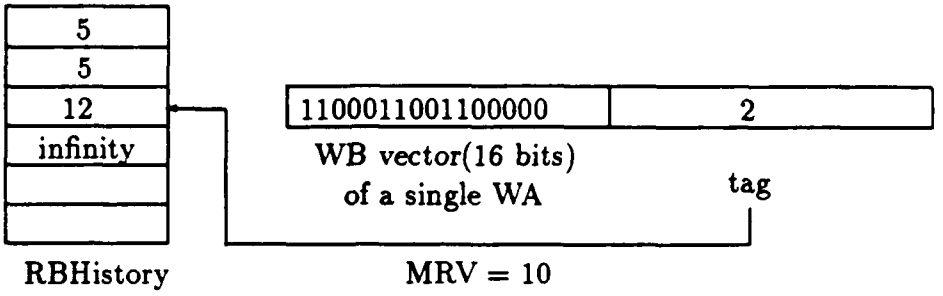
1. Define an array of values $RBH[t]$ such that $RBH[t]$ indicates the deepest rollback that has occurred since time $t$.

2. Whenever a block of written bits are written to memory (normally, 16 written bits will be written at one time), store a timestamp $ts$ with the written bits indicating the current time.

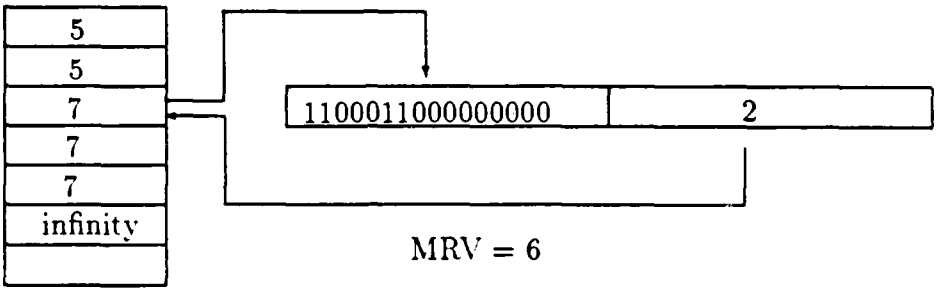
3. Whenever the written bits are read from memory, read the timestamp $ts$ that is stored with them, and clear all bits corresponding to frames that are newer than (greater than) $RBH[ts]$.

Although this approach efficiently implements the bit clearing operation, it has a serious flaw: the size of the $RBH$ array must have an infinite number of entries because the index $t$ is a continuous quantity. This problem is resolved by observing that $RBH[t]$ (the deepest rollback since time $t$) is identical to $RBH[t + \Delta t]$ if no rollbacks occurred between $t$ and $t+\Delta t$. Therefore, the $RBH$ entries corresponding to times between two consecutive rollbacks can be represented by a *single* $RBH$ entry. This is equivalent to saying that "time," from the perspective of RB histories, is measured by the number of rollbacks that have occurred since the computation began. Each rollback increases $RBH$ time by one unit. The timestamp, described above, is simply "the number of rollbacks that have occurred since the computation began." One need only maintain a counter that is incremented each time the process is rolled back, and use this counter to generate timestamps (called "tag", used as an index to RBHistory stack entries) when written bits are written to memory. An example of using this lazy approach is shown in figure 3.5.

With the above modification, the lazy approach to clearing written bits can be implemented very efficiently. The only question that remains is maintaining




(a) Rather than reset the written bits on rollback,
the WB information of a single WA and its tag are written into memory




(b) When reloading the line, the WB vector is read and reset.
Bits in frames newer than 7 are cleared.

Figure 3.5. Example: lazy approach of clearing the WB by using tag

the *RBH* array. The *RBH* array can be viewed as a stack, with a new element pushed onto the stack whenever a rollback occurs. Stack elements are *never* popped from the top of the stack, however, the oldest entries at the bottom of the stack may be garbage collected. Technically, the *RBH* mechanism is actually a FIFO queue, but we shall refer to it as a stack to facilitate the presentation. It is actually implemented as a circular buffer, so all of the arithmetic described below is implicitly modulo arithmetic.

$RBH[i]$ indicates the destination frame number of the "deepest" rollback that has occurred *after* the *ith* rollback ($i+1$, $i+2$, etc.), or equivalently, after the stack element was created. The top of stack element always contains the value INFINITY

```
RBH-UPDATE(dst)
     i := CRBI;
     while (dst < RBH[i])
          RBH[i] := dst;
          i := i-1;
          end-while
end RBH-UPDATE;
```

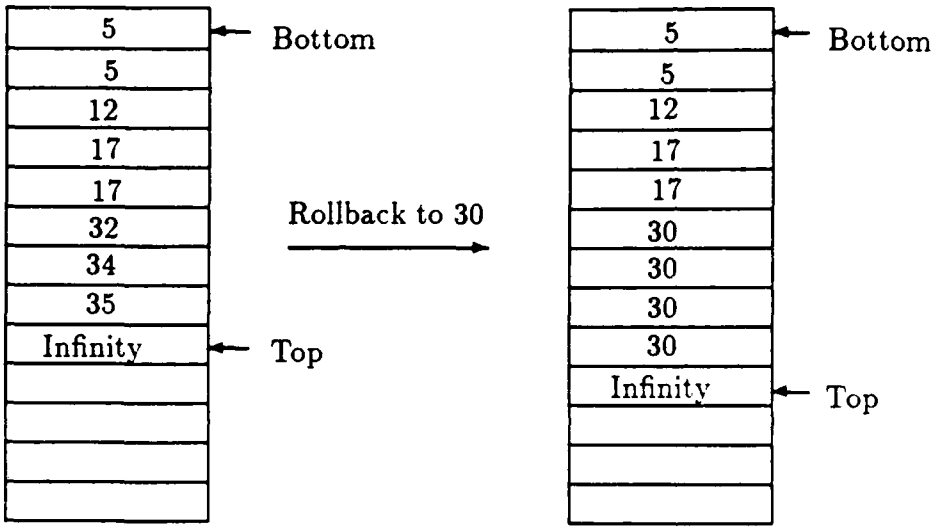
Figure 3.6. Update operation for RBH stack for rollback to frame dst.

because no subsequent rollbacks have yet occurred. When a rollback occurs, one must update the $RBH$ stack — if the destination *dst* of the current rollback is deeper than (less than) $RBH[i]$, then *dst* should be written into $RBH[i]$. At first glance, this would imply the entire $RBH$ stack must be examined on each rollback. Fortunately, this is not the case.

It is easy to see that the condition $RBH[i] \leq RBH[i+1]$ must always be true — the deepest rollback since time $i$ must clearly be at least as deep as the deepest rollback since time $i+1$.[1] Therefore, if rollback history entries are updated from the most recent to the oldest, we can stop the updating process *as soon as a rollback history entry is encountered with a rollback as deep or deeper than the destination of the current rollback.* If the rollback is relatively short (i.e., if there is temporal locality), very few rollback history entries will have to updated. The update procedure for the RBH stack is shown in figure 3.6. An example of the RB history stack before and after a rollback operation is illustrated in figure 3.7.

The update procedure may be efficiently implemented by buffering the top portion of the stack in a special custom memory with embedded comparison logic to update stack entries in parallel whenever a rollback occurs. The remainder of the

[1] Another way of seeing this is to observe that the set of rollbacks since $i+1$ are a subset of those since $i$.




(a) Before rollback.

(b) After rollback.

(1) new entry pushed.

(2) older entries updated.

Figure 3.7. Updating RBH entries on rollbacks.

stack is stored in conventional RAM. Only long rollbacks will require updates to the rollback history elements that are stored in RAM, so the entire rollback history stack can usually be updated very rapidly. After performing extensive simulations of the rollback chip (described later), we have never observed more than ten entries of the RBH stack updated on a single rollback — on average, only two to three entries are updated (one entry, the top of stack, is *always* updated on each rollback). Therefore, by buffering only a modest number of RBH entries in the custom chip (say 16), one would expect that the entire stack can be updated in a single clock cycle.

Garbage collecting the rollback history (from the bottom of the stack) is straight-

forward. A variable called $TAGBOUND[wa]$ is associated with working area $wa$ that holds the pointer to the top of the rollback history stack ($CRBI$ or current roll back index) when $wa$ was *first* created by a MARK operation (i.e., recreation following subsequent rollbacks is ignored). $TAGBOUND[wa]$ is a lower bound of any tag (timestamp) written into working area $wa$. Consider two consecutive working areas, $wa$ and $wa+1$, that are currently in use. Because $wa+1$ must have been created *after* $wa$ was created, and CRBI is always increasing in value (in the modulo sense), then $TAGBOUND[wa] \leq TAGBOUND[wa+1]$. It immediately follows that $TAGBOUND[OWA]$ is a bound on the smallest tag in use by any working area. Thus, when working area $wa$ is fossil collected, rollback history entries up to, but *not* including $TAGBOUND[wa+1]$ may be reclaimed.

## 3.5 Fossil Collection

The ADVANCE operation triggers fossil collection of storage that is no longer needed. Irrevocable operations, such as I/O, are also performed during fossil collection. For example, each output device can be implemented as a separate logical process. When there is output required, a timestamped message is sent to that process; physical output is generated after GVT exceeds the timestamp on that message. These mechanisms are independent of RBC operation (which only manages the state queue), so they will not be discussed further.

It is convenient (and more efficient) to process an entire working area at a time rather than on a frame-by-frame basis. One could, in fact, garbage collect *several* working areas at one time if even greater efficiency is desired, although this will complicate the mechanism somewhat. Data copying is required on storage reclamation because of the seldom written data problem mentioned earlier.

There may be much data copying (to the archive frame) required if many data values have been written into the fossil collected working area. This would degrade

performance significantly if the CPU were forced to wait until the data copying were completed. This problem is avoided by performing the ADVANCE operation in parallel with other RBC operations.

The ADVANCE operation has only a minor effect on the RB cache if a write-through policy is used. If the ADVANCE operation fossil collects data that is stored in some entry of the cache, the MRV field of that cache entry will become out of date. However, even if this MRV information is left out of date, the cache will still operate correctly because the MRV information is only used during the invalidation operation when a rollback occurs; the worst that could happen is a cache entry might be accidentally invalidated by a rollback. Accidental invalidation might degrade performance slightly, but does not compromise correctness. If desired, accidental invalidation could be avoided by resetting the MRV field to a special state that cannot be invalidated by rollback whenever the MRV frame is fossil collected.

As mentioned earlier, if a copy-back policy is used, entries with set DirtyWB and/or DirtyData bits must be flushed to the memory before the fossil collection because the latter uses only information stored in memory to reduce the cache contention.

Because the ADVANCE operation proceeds in parallel with other RBC operations, some care must be taken to avoid race conditions. In particular, the ADVANCE operation copies lines from working areas into the archive frame concurrently with memory operations that may also access the archive frame. Race conditions can be avoided by simply delaying the increment of the OMF register until all data copying is completed. This avoids races because: if the most recent version of the line is in the archive frame, there are no *set* written in the working area that is being garbage collected, so the ADVANCE operation performs no data copying and no race condition can occur. On the other hand, if the most recent

version is *not* in the archive frame, READ and WRITE operations will never access the archive because they only reference the most recent version of data. Again, the correct MRV information will be accessed, so no race condition is possible.

Finally, a simple optimization can be used to reduce the amount of data copied to the archive frame. If there is at least one set written bit in a frame that is newer than the working area being garbage collected but still at least as old as the value of the OMF after the ADVANCE is complete, then the data need not be copied to the archive frame. This requires the ADVANCE operation to read some additional written bits to determine if it need not copy the data, however this is less expensive than copying the line.

## 3.6 Dynamic Memory Allocation

Allocating memory to every line of every frame of the mark frame stack will require a substantial amount of memory. For example, if a process has 4M bytes of VCM, and the stack contains 256 frames, 1G bytes of memory are required for each process. If only a small number of state variables are changed between MARK operations, most of the memory space remains unused and is wasted.

Dynamic memory allocation using *demand paging* techniques are used to address this problem. Here, the "virtual" address is a pointer into the mark frame stack, (obtained by concatenating the address generated by the CPU with a frame number) and the physical address refers to main memory. A page table is associated with each VCM. Each page table entry contains a presence bit that is set if physical memory has been allocated for the page, and reset otherwise. If the presence bit is set, the page table entry also contains a pointer to the page. A new page is allocated on the first write into that page. Unused pages are maintained in a free list.

Because of the large size of the VCM, the page table associated with each working area may require too many entries. Techniques using multiple levels of page table can be used to reduce the memory required for page tables.

## 3.7 Multiple Processes per Processor

The expense of context switches between processes is reduced by the PID field included in the RB cache. The associative searches are performed on the PID field as well as the line number field. This allows the cache to hold data from different processes simultaneously. Similar techniques are used in translation lookaside buffers in conventional MMUs. "Synonym" problem [26] associated with traditional virtual memory caches will not arise in the RBC because Time Warp algorithms exclude shared memory between simulation processes. Finally, other process specific state (e.g. CMF/OMF, the rollback history stack and TAGBOUND) must also be swapped on context switches, or techniques using multiple register banks must be used.

# CHAPTER 4

# PERFORMANCE

The overhead incurred by the rollback chip has been evaluated through extensive simulation studies of the RBC mechanisms. In particular, much of the evaluation has been focused on the performance of the RB cache. The MARK and ROLLBACK operations can be performed in constant time (only a few clock cycles are required), as will be discussed later.

Rather than compare the RBC to a hopelessly inefficient software mechanism, we compare it to a comparable conventional cache memory with *no state saving overhead.* This will enable quantitative measurement of the cost incurred by the RBC to implement state saving.

## 4.1 The Simulation Model

A simulator has been developed for the rollback chip. Partial validation of the simulator was obtained by comparing its operation to an independently developed simulator that implements a simple, brute force version of the RBC algorithm using copying. The two simulators were exercised and compared over several million operations, and found to yield identical results (i.e., corresponding read operation returned the same value).

The RBC operation requests from the CPU are generated stochastically from a multinomial distribution – a fixed probability is assigned to each of the five operations (RESET is only generated once at the beginning of the simulation). These requests form the workload presented to the RBC. Traces from an existing

Time Warp system were not readily available. However, even if such traces could be obtained, they would not provide a true characterization of the expected RBC workload because the frequency and distance of rollback operations are timing dependent, and would not reflect operation using the RBC. On the other hand, using a stochastic workload generator allows generation of a wide range of workloads in an easily controlled fashion.

Addresses for READ and WRITE operations are generated stochastically from a normally distributed random variable. The locality of the address trace is controlled by adjusting the variance of this distribution, but remains fixed in any single simulation experiment. Also, the mean of the normal distribution is periodically changed (within a single simulation run) to model phase changes in the computation.

As described earlier, the operation of the RBC is such that READ and WRITE operations that "hit" can be expected to require the same amount of time as a hit in a conventional cache. Although a write hit in the RBC may generate additional memory traffic (e.g. to update the written bit array), the CPU need not wait for these memory accesses to complete. Further, because instruction references and many data references do not access version controlled memory, the RBC is usually afforded some time to complete these memory accesses before a new RBC operation is initiated.

Therefore, two important questions to be asked are (1) can the RBC achieve hit rates comparable to conventional caches, (2) is the miss penalty in the RBC significantly larger? A lower hit rate would be expected in the RBC because ROLLBACK operations usually invalidate some cache entries. The miss penalty is larger because written bits must be searched to locate the MRV frame. To allow fair comparisons of hit rate, a "comparable" conventional cache is defined as one that is identical to the RBC but ignores all RBC operations except READs and WRITEs.

Parameters that are used to control the simulator can be characterized into two categories: (1) workload parameters, and (2) cache design parameters. The former includes:

- the size of version controlled memory. This was fixed at 4096 lines.
- the locality of the address trace. The locality was fixed in each experiment; four different degrees of locality were examined.
- the number of reads and writes between MARK operations (the computation granularity). This was fixed at 20 to correspond to small grained events, and 200 to model large grained events.
- the frequency of WRITE operations relative to READs. This was fixed at 2 or 4 READs per WRITE in each experiment.
- the number of MARK operations between ROLLBACK operations (this affects the relative event rate, defined later). This was fixed at 2, 8, or 16 in each experiment.
- the frequency of rollbacks and distance of rollbacks. Within each relative event rate, the rollback distance was fixed at 2, or 8 in each experiment.
- the frequency at which the mean of the address distribution changes. This was fixed at 3, 7.5, or 15 in each experiment.
- the average number of mark frames in use. This was fixed at 30 frames, unless indicated otherwise.

The cache design parameters include:

- the size of the cache, this was set at 256 cache entries.
- the cache organization (direct mapped, set associative, or fully associative),

- the cache design (design 1 using write-through vs. design 2 using copy-back).
- the size of the mark frame stack. This was fixed at 256 frames.
- the use of the LastWA optimization for cache misses.

## 4.2 Simulation Methodology

Exhaustive simulations using all possible combinations of the above parameters is impractical. The following approach was taken to evaluate the effect of each of these parameters on performance:

1. select a "base" set of cache parameters for the cache design and a set of "typical" workloads,
2. run the simulator across a wide range of workloads with the base cache design parameters to evaluate the performance of this particular design, and
3. run the simulator across different cache design parameters, using the typical parameter settings of the workload to evaluate the effect of alternative designs.

The base selection for simulating the RB cache is two-way associative, write-through, and size of 256 entries. Typical setting for workload parameters includes 7.5 for locality changes, and 4 for READ/WRITE ratio; other workload parameters (i.e., address locality, relative event rate, and rollback distance) are as described earlier.

## 4.3 Relative Event Rate

Rollbacks reduce the RB cache hit rate by invalidating entries. The more frequently rollback occurs, the more often entries are invalidated. Similarly, the longer the rollback, the more entries invalidated by each rollback. However, simulation

programs can never have rollback operation that are *both* frequent (relative to the frequency of MARK operation) *and* long. Let $F_{MK}$ and $F_{RB}$ denote the frequencies of MARK and ROLLBACK operations respectively, and $RB_{dist}$ be the average rollback distance. The quantity $F_{MK}/(F_{RB} * RB_{dist})$ indicates the net rate at which events are being processed. Referred to as the relative event rate, this quantity must be greater than one or else the computation is going backward! This is provably impossible in Time Warp. An application program based on Time Warp with larger event rates normally results in a better hit rate. As the event rate approaches infinity, rollbacks are less frequent and/or shorter, and the hit rate will approach that of a conventional cache. On the other hand, programs with poor event rate (e.g. nine steps backward per ten steps forward) can be expected to suffer increased hit rate degradations relative to the conventional cache because rollbacks may invalidate cache entries.

## 4.4 Hit Rate

The hit rate of the RB cache is defined as the ratio between the number of the references to version controlled memory variables which are found in the RB cache and the total number of the references to the VCM variables. Here, the important measurement is not the absolute hit rate (any value can be obtained by varying the locality of the address trace), but rather the hit rate degradation when compared to that of a comparable conventional cache (i.e., one with no entries invalidated by rollback operations).

### 4.4.1 The Affecting Factors

A series of simulation experiments were performed to evaluate the amount of hit rate degradation using the RBC under various workloads. A complete table

of these results can be found in an Appendix. The effects of different workload parameters on performance are discussed below:

**Locality of address traces:** The absolute hit rate of the conventional cache is controlled by varying the locality of the address trace, which is in turn controlled by the deviation of the normally distributed random variable used to generate addresses. The smaller the standard deviation, the higher the locality, and the higher the hit rate. Experiments were conducted in which the hit rate ranged from 70% to, nearly, 100%. Hit rates in modern conventional caches are normally 90% or higher, and often greater than 99%.

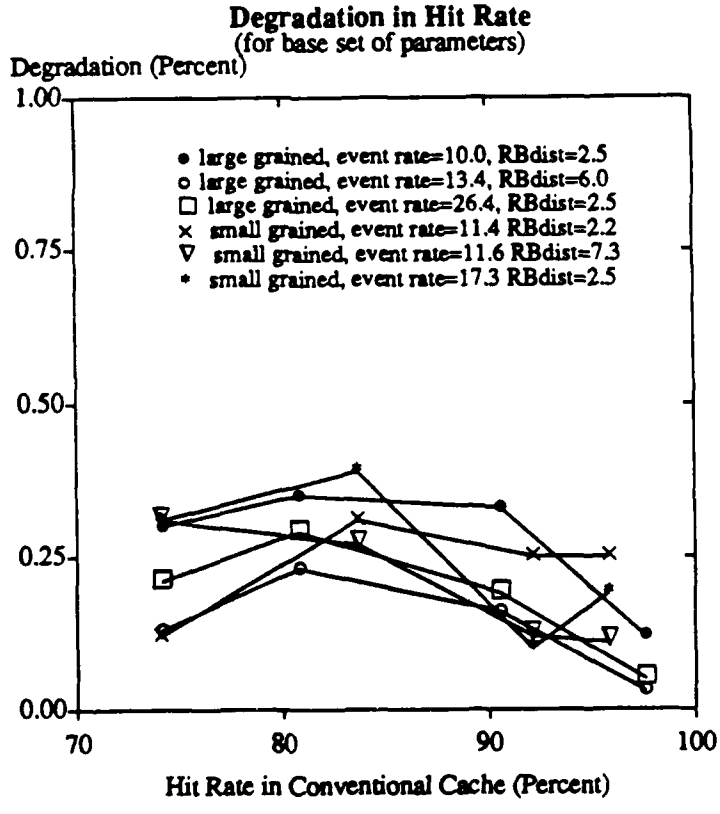



Figure 4.1. The hit rate degradation with base set of parameters

The hit rate degradation for different absolute hit rates for the base set of parameters are shown in figure 4.1. The hit rate degradation tends to decrease as the *absolute* hit rate increases, especially at very high hit rates. This is because as locality is improved, fewer cache entries tend to be invalidated by rollback; for example, if all memory references were to a *single* memory address, the rollback invalidation operation would only invalidate at most a single entry of the RB cache. This effect is less significant for lower hit rates because the size of the cache then becomes a significant factor; if the cache is too small, the replacement policy will tend to delete entries before the rollback has a chance to invalidate them.

**Rollback distance and event rate:** As discussed earlier, lower event rates and/or shorter rollback distances tend to increase the degradation in hit rate. Quantitatively, the effect of these parameters is shown in figure 4.1. The situations where the RB cache experiences the most degradation corresponds to those cases where the event rate is very poor. However, in these situation, the Time Warp program is thrashing, so performance of the RB cache is a mute point. Therefore, only the situations corresponding to "reasonable" event rates (e.g., 8 or 16) are of practical interest. In these cases, the degradation is less than 0.5% in most cases. Shorter rollback distances (within a fixed event rate) tend to increase the degradation (0.1%-0.3%), especially for the large grained computations because more recently written entries are invalidated.

**Frequency of locality changes:** A locality change results in a flurry of misses until the new working set is loaded into the cache. Hit rate degradation for locality changes every 3, 7.5, and 15 events are shown in figure 4.2.

As can be seen from the results, less frequent changes in locality will cause greater degradation in the hit rate. For the parameters used in these ex-

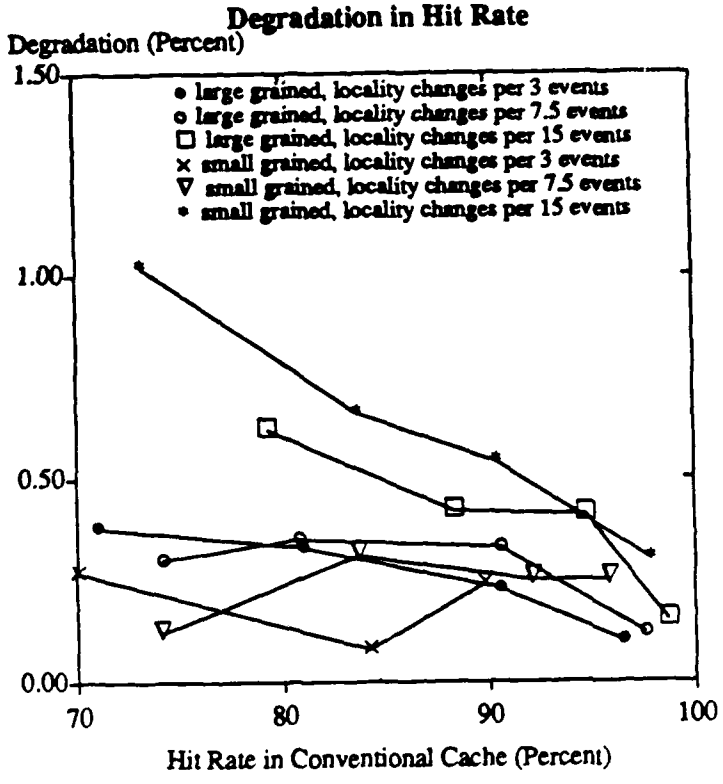



Figure 4.2. The effect of the frequency of locality change on hit rate degradation

periments, hit rate degradation increased by up to 1.5%. This is because if the frequency of locality changes is lower, the ROLLBACK operations tent to invalidate entries that are more likely to be referenced again in the near future.

**Granularity of computation:** Programs with large grained events, i.e. more accesses to version controlled memory between MARK operations, are expected to have higher absolute hit rates because more read and write operations occurs between rollbacks assuming a fixed event rate and rollback distance. Results of experiments, for small grained events (20 READs and WRITEs between MARK operations), and for large grained (200 READs and WRITEs between MARK operations) are shown in figure 4.3, and 4.4 respectively.

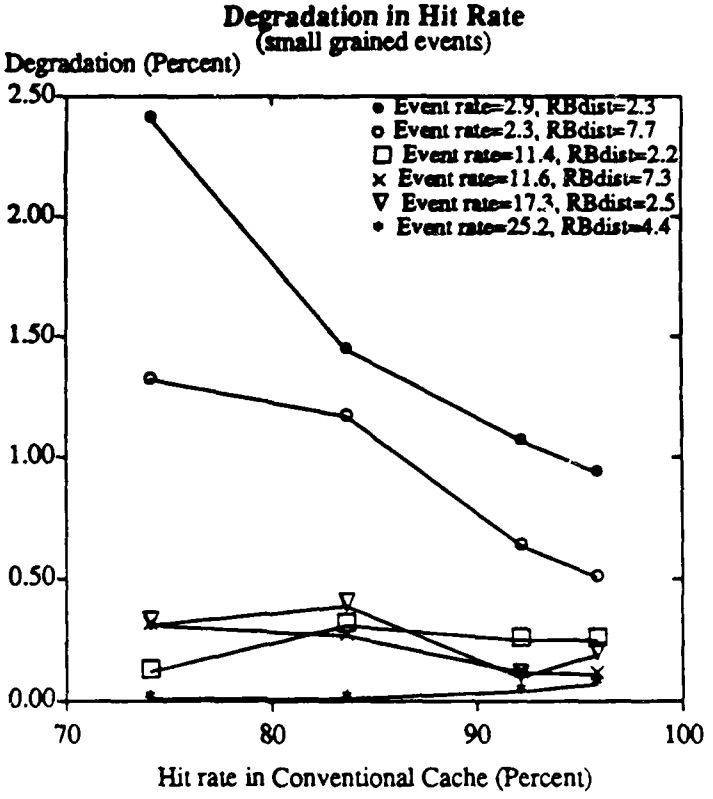



Figure 4.3. Hit rate degradation for small grained computation event

In each graph the degradation in hit rate is plotted as a function of the hit rate in the conventional cache, which in turn is controlled by adjusting the variance in the probability distribution used to generate the address trace. As can be seen, the degradation in hit rate using the RB cache varies from less than 0.01% to as much as 2.41%.

**Reference and modification ratio:** The invalidation of RB cache entries is based on the comparison of the rollback destination (new CMF) and MRV cached in each entry. Thus, entries that are written frequently tend to have more recent MRVs and are more likely to be invalidated by rollbacks. One would expect higher degradation as the frequency of the WRITE operations relative

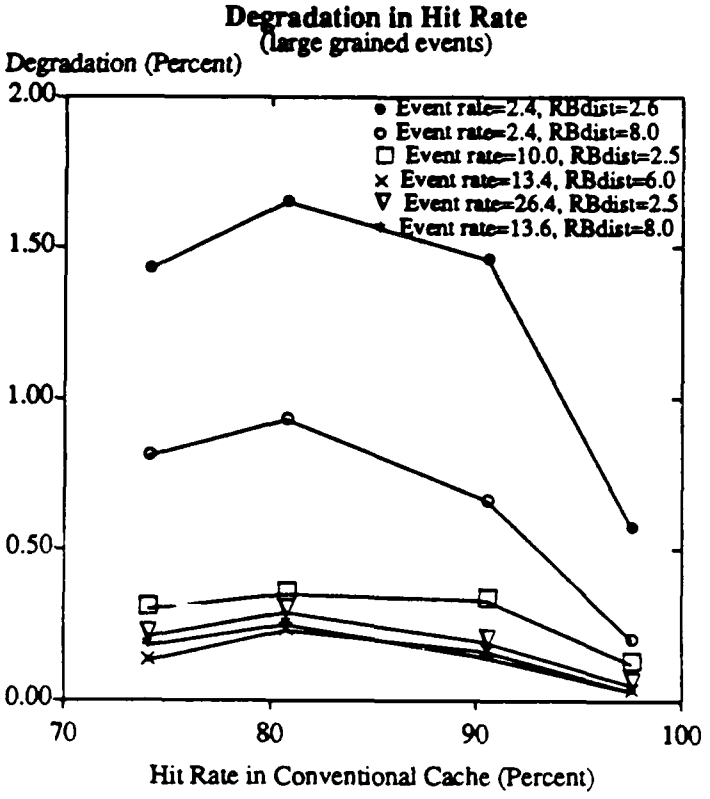



Figure 4.4. Hit rate degradation for large grained computation events

to READs is increased. Results of experiments in which the 66% and 80% of memory accesses are reads are shown in figure 4.5. For large grained computation, the degradation is increased by 0.1% to 0.2% when the percentage of write operations increased from 20% to 33%. An increase up to 1.5% in degradation can be expected for small grained computation.

Another series of simulation experiments were conducted that evaluate the performance of different cache design strategies (a typical workload with locality changing every 7.5 events, and four READs per WRITE operation is assumed):

**Cache organization:** Results of different RB cache designs using direct address, two-way associative, and fully associative cache organizations are shown in

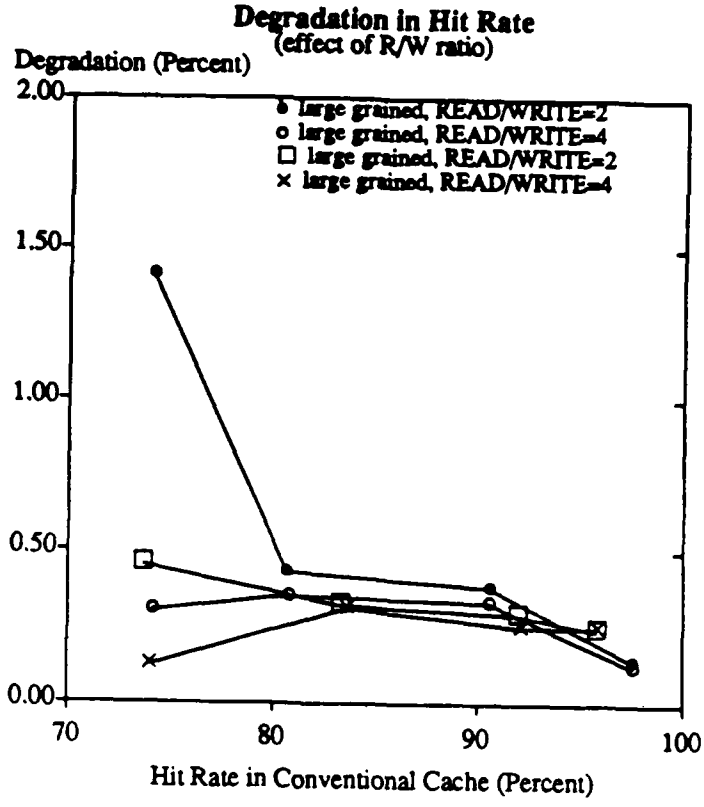



Figure 4.5. The effect of READ/WRITE ratio on hit rate degradation

figure 4.6. When the absolute hit rate is low (e.g. 70%), the degradation of the fully associative organization was observed to be smaller than that of the others. However, in those experiments with low event rates (e.g. 2). the degradation using the fully associative approach increases significantly as the absolute hit rate increases, and eventually exceeds that of the other organizations (see the Appendix for numerical values).

The reason for this behavior has to do with the fact that rollbacks always invalidate cache entries corresponding to the most recent write operations. A higher degree of associativity implies that a larger number of the most recently written lines are kept in the cache (assuming an LRU replacement policy is

used), so more cache entries will be invalidated by rollback operations when the absolute hit rate is high. An inspection of the number of the RB entries that were invalidated in the simulation experiments described above supports this explaination.

In general, with a lower degree of associativity, more entries corresponding to *non-active* sets (those which have not been referenced recently, but have not yet been deleted from the cache by the replacement policy), are likely to exist in the cache than in the fully associative organization. If the reexecution of the computation after the rollback is similar to that of the original, the probability that these entries will be referenced again by the reexecution phase is higher than if the rollback had not occurred. Because the caches with lower degree of associativity are more likely to keep these lines in the cache, they can be expected to suffer fewer misses immediately following the rollback. This effect might also be obtained by using a different cache replacement policy.

For those experiments with reasonable event rates (8, 16), the degradation was observed to be at most 1.74%. Further, the trend in cache design today is toward large caches with a small degree of associativity (i.e., direct address caches) to simplify the cache circuitry and thereby reduce cache access time. In this context, we expect that hit rate degradation using the RB cache will be very small, much less than 1interest.

**Write policy:** Because a copy-back RB cache contains a WB field to cache more versions, the degradation of it is smaller than that of a write-through RB cache[1]. However, the improvement in degradation is small when a (much

[1]The comparison in write policy is done for fully associative caches because the original simulator developed for the copy-back cache only implements fully associative searches.

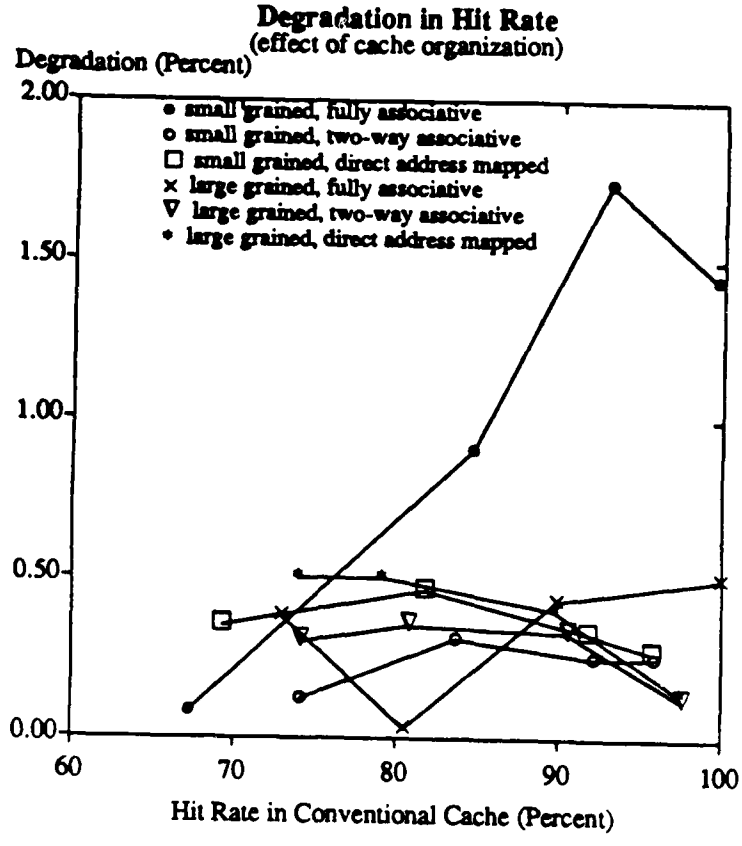



Figure 4.6. The effect of cache organization on hit rate degradation

more complicated) copy-back RB cache is used. Therefore, the write-through policy is recommended in the RB cache.

## 4.5 Miss Penalty

Even if high hit rates can be obtained, overall performance would be disappointing if the penalty of each miss was very large. Overheads incurred by the RB cache on a read or write miss that are *not* incurred in a conventional cache include: (1) the written bits and the associated RBH tag must be read, (2) the RBH stack must be read, and the appropriate written bits cleared, and (3) the page table entry must be read to locate the line data. (1) may be incurred many times on a single

miss if the RB cache must search a long distance to locate the MRV frame, and is the principal point of concern. (2) is also required on each iteration of the search, but it incurs a performance penalty on only the first iteration if the hardware is pipelined. The page table reference (3) is only required once at the end of the search. By using a translation lookaside buffer and overlapping access to it with access to the written bit memory, one can eliminate performance degradation for address translations in most situations.

Two search strategies were proposed in the RB cache design. The original approach begins the MRV search from the CMF frame. An optimization was proposed that begins searching from the "last written" working area (LastWA). This latter approach necessitates an additional memory reference on each miss to read the LastWA information.

### 4.5.1 The Affecting Factors

Miss penalties for both the optimized and unoptimized search strategies were compared to evaluate the usefulness of the optimization. The factors that have the greatest effect on the miss penalty include:

**Active frames:** An important factor that affects the miss penalty is the number of the active frames, i.e. all the frames between the CMF and the OMF. This number is especially important for the unoptimized search strategy, because it places an upper bound on the length of the required search; for example, application programs with only one active frame (i.e. CMF=OMF) need only check the current frame to find the most recent version.

Results of simulations to evaluate the miss penalty are shown in figure 4.7 for both the optimized and unoptimized strategies. Numerical data is included in an appendix. The search distance (number of blocks of written bits that must be read to locate the working area with the MRV frame; recall that 16

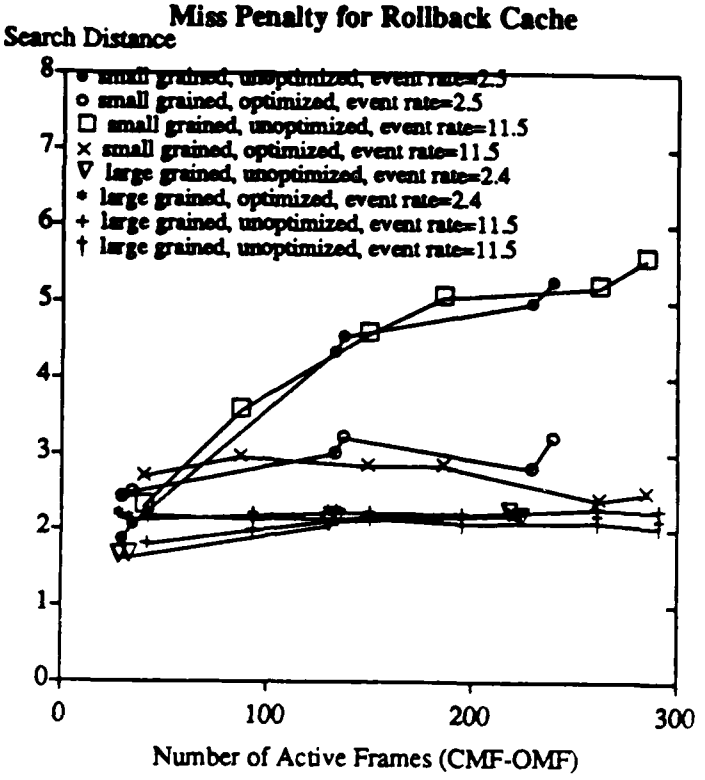



Figure 4.7. The effect of the number of frames in use on miss penalties

written bits are read on each memory reference) is plotted as a function of the number of active frames ($CMF - OMF$). The figures for the optimized version *include* the additional memory reference to access LastWA information so that fair comparison can be made.

**Locality of address traces:** The address distribution will also affect the search length because it (the search length) is proportional to the amount of time that has passed since the data was last written. For example, one would expect that seldom written data requires a longer search distances than frequently written data. These results are shown in figure 4.8. For large grained computation, an average increase of 0.3-0.7 blocks in search length is required to locate the

MRV frame as the absolute hit rate decreases from 97% to 74%. For small grained computations, as the mark frame stack grows, an increase of up to 4.5 blocks was observed.

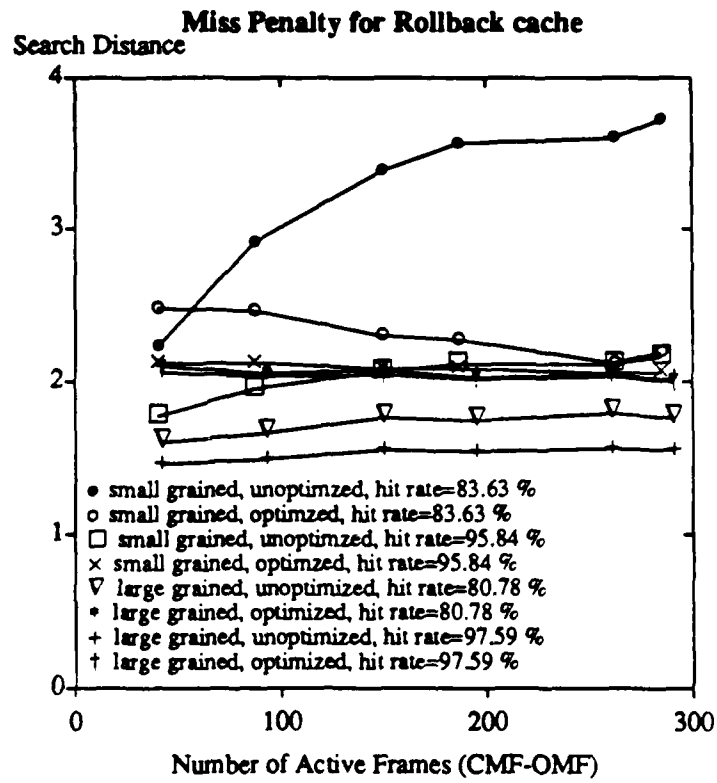



Figure 4.8. The effect of locality of address traces on miss penalties

**Reference and modification ratio:** Data that is modified frequently will be expected to have a shorter search distance, especially for the unoptimized strategy. As shown in figure 4.9, the average increase in search length was observed to be less than 0.1 blocks in most cases for large grained computations, and 0.1-0.5 blocks for small grained computations when the READ/WRITE ratio increases from 2 to 4.

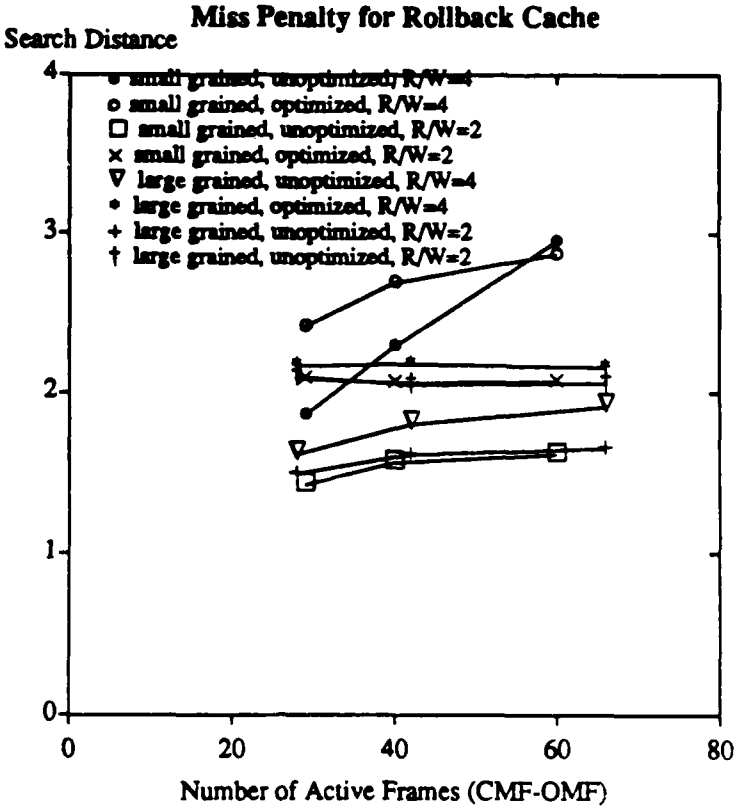



Figure 4.9. The effect of READ/WRITE ratio on miss penalties

**Rollback distance:** The rollback distribution will also impact the search length, particularly for the optimized strategy – if no rollbacks occurred, then LastWA will always point to the working area containing the MRV frame. As seen in figure 4.10, the average increase in search length is less than 0.1 block in most cases for large grained computation, and was 0.1-1.3 blocks for small grained computations.

The simulation results are encouraging in that even for long $CMF - OMF$ distances, LastWA in the optimized version usually points directly at the working area containing the MRV information, or close to it. However, for a small number of active frames, the unoptimized scheme is somewhat better because the LastWA

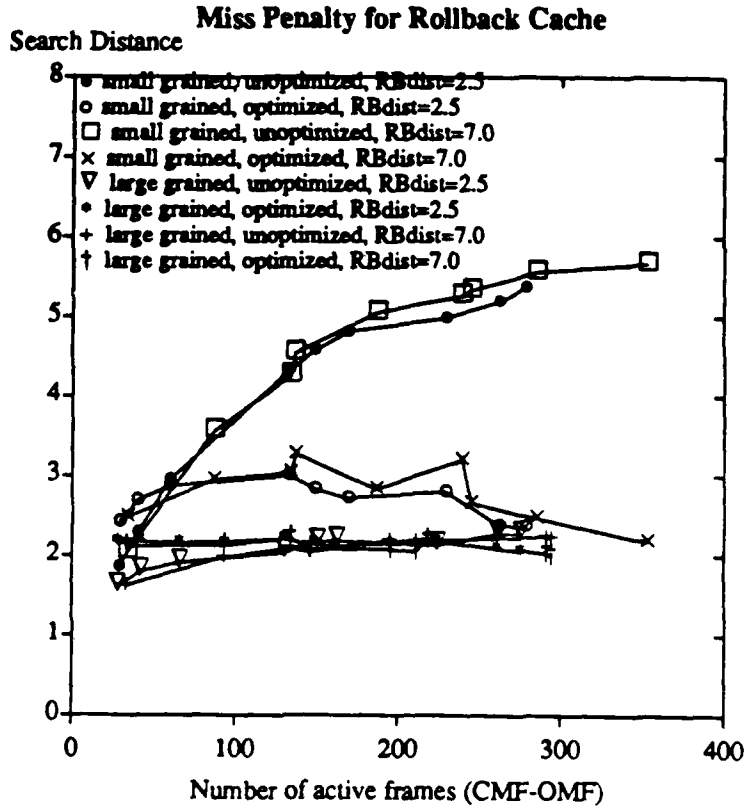



Figure 4.10. The effect of rollback distances on miss penalties

value need not be read. Similarly, if large grained events occur that typically modify a large portion of the process state, search distances are also short, so the unoptimized approach is preferred. These results indicate that the unoptimized strategy is adequate, and in fact preferred, in many situations that are expected to arise in practice. If large stacks and small grained events may arise, an adaptive strategy could be used in which the optimization is enabled if it appears that long searches are taking place.

## 4.6 Performance of the Rollback Operation

Two aspects of the rollback history are of particular interest: the number of the entries that must be updated on each rollback and the size of the RBHistory stack. To economize on circuitry, only the top portion of the RBH stack is maintained in the RBC; the rest of it is stored in memory. Those entries stored in the RBC can be updated in parallel, while those stored in memory must updated sequentially using RBC microcode. Therefore, the number of entries updated on a rollback gives an indication of the number of entries that should be maintained in the RBC.

Experiments were performed across a wide range of rollback scenarios. The results are shown in figure 4.11. In these simulation experiments, the average number of updated entries is less than 2 on each rollback. A minimum of 1 entry is always updated, and typically only 2 to 3 entries are updated on each rollback: these results indicate that only a few entries need to be buffered in the RBC. Alternatively, it is not unreasonable not to buffer any entries in the RBC, and perform the entire update operation in microcode. These experiments also indicate that the size of the rollback history stack is usually only a few tens (about 20 to 40) of entries.

## 4.7 Overall Performance

The simulation results indicate that when compared to a conventional cache that does not perform any state saving functions, the RBC suffered a loss of 0.01 to 2.41 percent in hit rate (for those with reasonable event rates, such as 8, 16). and required 1.5 to 3.0 additional memory access on READ and WRITE misses to locate the MRV frame. Of course, actual hit rate degradation and miss penalties are highly application dependent; nevertheless, the simulation results give an indication of expected performance.

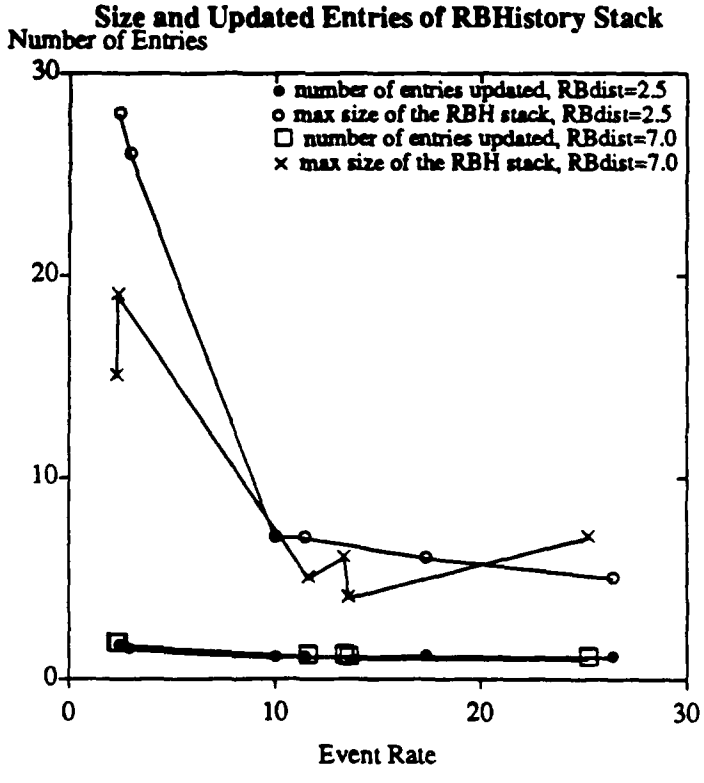



Figure 4.11. The size of the RBH stack and the number of updated entries on each rollback

The average memory access time for a cache memory system is $P_{hit}T_{cache} + (1.0 - P_{hit})T_{memory}$ where $P_{hit}$ is the probability of a cache hit, and $T_{cache}$ and $T_{memory}$ are the access time to the cache and main memory, respectively. For example, consider a design based on a 30 MHz INMOS Transputer, e.g., the IMS-T800 [16]. Assume cache hits can be processed without introducing wait states (for the transputer, this implies an access time $T_{cache}$ of 100 nanoseconds.). Assume references to main memory require 200 nanoseconds ($T_{memory}$), and misses in the RB Cache incur an additional 200 nanosecond penalty. The written bits and tags are assumed to be stored in a dedicated, fast memory to prevent the miss penalty from becoming excessively large. From figure 4.3, it can be seen that for an event rate of 11.4, average rollback distance of 2.2, and hit rate of 95.84%, the degradation in hit rate is 0.25%. The average memory access time for the cache is 104.16 nanoseconds $(0.9584 * 100 + (1 - 0.9584) * 200)$. The RBC has an average memory access time 113.23 nanoseconds $(0.9559 * 100 + (1 - 0.9559) * (200 + 200))$. This yields an overall increase in the average memory access time of 8.7%.

Further, it should be pointed out that most memory references do not reference version controlled memory; instruction references, accesses to local variables that do not persist from one event to the next, and code associated with the Time Warp mechanism itself (e.g., for manipulation of input queues; these references constitute a very significant portion of the computation for fine grained events) bypass the rollback chip completely. When taking this into account, overall performance using the RBC will be virtually indistinguishable from that of a CPU with a conventional cache. For instance, if 10% of the memory references access version controlled memory, then the overall cost of state saving in the rollback chip using the parameters listed above is only a 0.87% degradation in performance.

Repeating this calculation for the remaining data points in figures 4.3 and 4.4 yields the curves shown in figure 4.12. The cost of state saving and rollback using

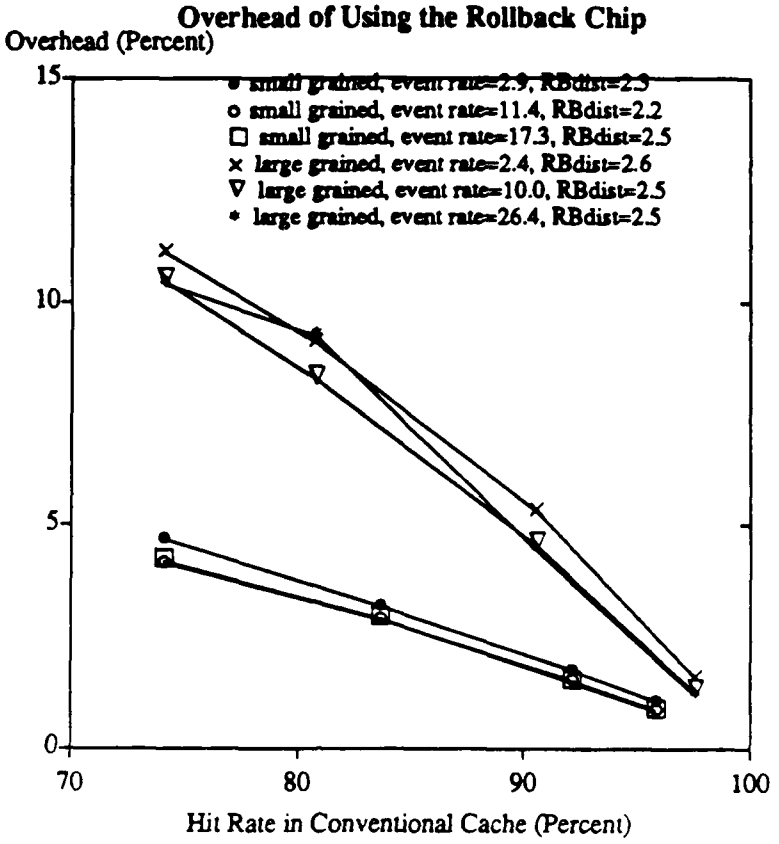



Figure 4.12. Overall degradation of using the RBC

the RBC is plotted as a function of the hit rate in the conventional cache. As before, $T_{cache}$ is assumed to be 100 nanoseconds, $T_{memory}$ is 200 nanoseconds, and the additional miss penalty in the RBC is 200 nanoseconds. The curves for short rollback distances (averaging 2.3-2.5 events) are shown; those for longer distances are similar. The curves for small grained events assume 10% of the memory references access the RBC, while those for larger grained events assume 25% (a smaller percentage of references are due to Time Warp overhead as the granularity increases).

RBC performance improves as the hit rate in the cache improves because performance degradation in the RBC only occurs on misses. Further, as noted earlier, hit

rate degradation in the RB cache is diminished as the absolute hit rate improves. Today, conventional cache memory systems routinely achieve hit rates well above 90%. Therefore, it can be expected that the cost of state saving using the rollback chip will typically be only a few percent of processor performance. The RBC will further enhance system performance by performing memory reclamation in parallel with the CPU.

# CHAPTER 5

# IMPLMENTATION OF THE RBC

A block diagram of one possible implementation of a multicomputer node using the rollback chip is shown in figure 5.1. The CPU provides the computation power for the node and circuitry for interprocessor communications (possibly implemented

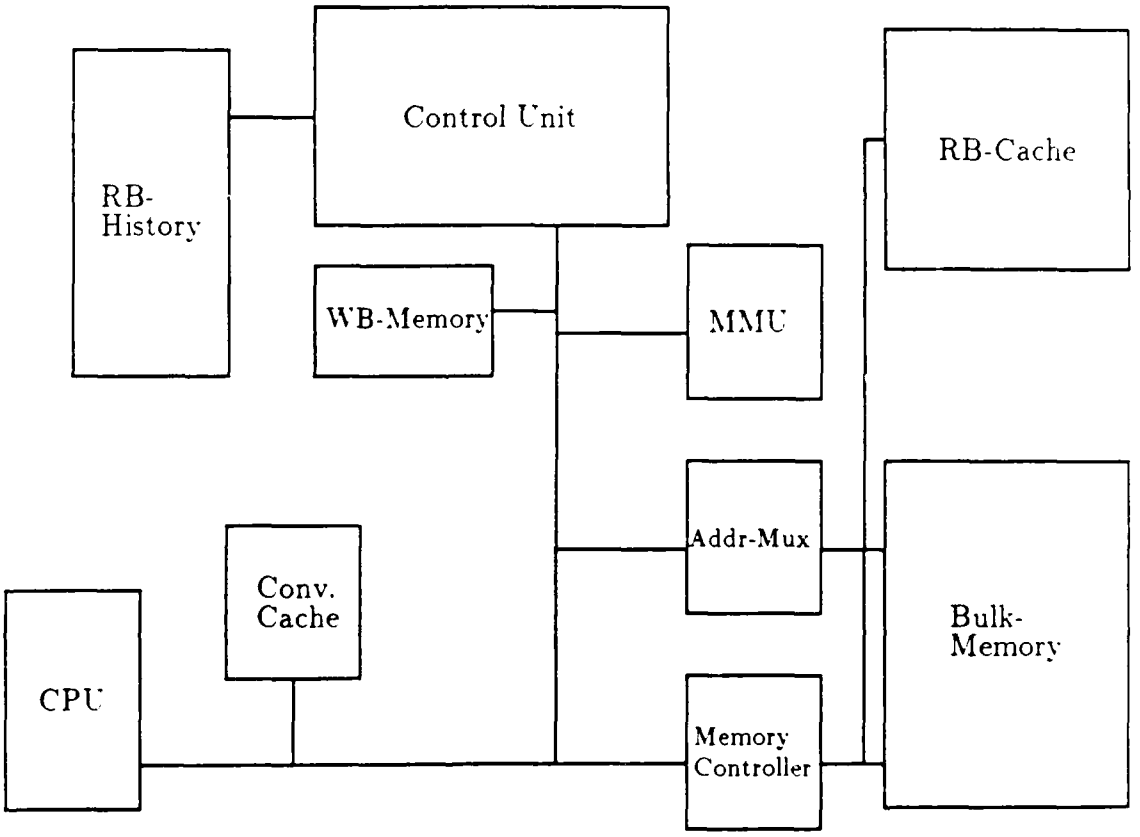



Figure 5.1. Configuration for each node of the simulation engine.

as a separate coprocessor). In this design, the CPU has a conventional cache associated with it to hold instructions and local (non-VCM) variables. This reduces memory contention with the RBC; the latter performs storage reclamation activities in parallel with the CPU. Alternatively, the mark frame stack could be stored in a separate physical memory from that holding instructions and local data. Bulk memory contains conventional dynamic RAM. The rollback chip hardware includes:

- *The control unit* (e.g., a microcode sequencer and ROM) to implement storage reclamation and other miscellaneous functions.
- *The RB cache*, including circuits for implementing the rollback invalidation and associative search functions.
- *Written bit memory*, implemented with fast static RAM. The RBH stack should also be stored here or in a separate high speed memory.
- *A memory management unit (MMU)* to implement the dynamic memory allocation scheme.
- One or more RBHistory units, which buffer the top portion of the RBH stack, and provide circuitry to allow rollback updates to be performed rapidly.

With the current technology, the RBC contains too much circuitry to be implemented as a single chip. However, excluding the static RAM portions of the chip, it could be implemented as a chip set of perhaps two or three VLSI components. Assuming circuits densities continue to grow as they have in the past, a single chip implementation of the RBC can be expected within a few years.

Existing commercial products can be used to implement the MMU. Similarly, the control unit could be easily implemented using off-the-shelf parts. Custom integrated circuits are required to perform the cache invalidation function and the RBHistory update.

## 5.1 The CPU

Pragmatic considerations make it highly desirable to use an off-the-shelf microprocessor in the simulation engine node. Many modern microprocessors contain an on-chip data cache. The RBC can be used with such components if appropriate precautions are taken. The most straightforward solution is to ensure that version control memory is never cached, or to simply disable the cache completely. Alternatively, the on-chip cache would have to be invalidated when rollback occurred. Further, if the microprocessor's cache use a copy-back policy, the cache would have to be flushed before each MARK operation. Similarly, for any processor that is used, one must ensure that internal processor registers are written to memory before each MARK operation if they must be restored on rollback.

Some microprocessors also contain an on-chip memory management unit. In this case, the RBC would have to reside between the MMU and physical memory, and receive physical memory addresses. The RBC assumes, however, that each version controlled memory occupies a contiguous portion of the address space. Therefore, the MMU address mapping would have to be controlled to ensure that this condition is not violated.

## 5.2 The Control Unit

The control unit is a microcoded engine that implements certain RBC operations. The control unit manages registers such as the OMF, CMF and CRBI registers. The registers must be swapped on context switches, or multiple banks of registers can be used. Also, the control unit implements the fossil collection and archiving of data. Functions whose performance are critical to the overall efficiency of the RBC, however, e.g., searches for MRV frames, are implemented with dedicated hardware.

## 5.3 The RB Cache

Associative searches are performed on the PID and Line fields of the RB cache. Conventional data cache hardware is used to implement this function. The MRV field of the cache must be compared with the destination frame of each rollback to determine whether the entry should be invalidated. The invalidations are performed on all entries simultaneously. The comparators embedded in the MRV and address (PID and Line) fields are enabled only when the Valid bit of that entry is set.

## 5.4 The RBHistory Stack

Circuitry similar to the invalidation circuit in the RB cache can be used to update the RBHistory stack. While the bulk of the RBHistory stack is stored in conventional static RAM, the top portion of the stack (TOS) is stored in a custom chip.

When a rollback occurs, each TOS register is replaced by the minimum of the value stored in that register and the destination of the rollback. This operation is performed in parallel by the comparison logic embedded in the TOS registers. If all of the TOS registers are modified, the control unit sequentially updates as much of the RBH stack as necessary. Finally, the oldest TOS register is then written to memory, and overwritten by the constant infinity. The simulation results reported earlier indicate that only a modest number of TOS registers are required (e.g., 8 or 16) to allow update operation to be confined to within the custom RBHistory circuit.

## 5.5 Overall Operation of the RBC

The CPU initiates the RESET, MARK, ROLLBACK, and ADVANCE operation by writing into the control registers. These operations invoke microcode sequences

that implement the operations. Details of the implementation of these operations are described in the code for the RBC simulator which is included as an appendix.

Memory READ and WRITE operations are implemented with dedicated circuitry rather than microcode to maximize performance. Each memory operation to VCM initiates an associative search in the cache. If a hit occurs, the data is either returned or overwritten as described earlier; otherwise, the WB memory is scanned to determine the MRV frame, and memory operations to the MRV data stored in bulk memory are performed.

# CHAPTER 6

# CONCLUSION

A special purpose component, the rollback chip, is proposed to offload the state saving and rollback overhead in the Time Warp parallel simulation algorithm. It is a key component of a special purpose, discrete event simulation engine based on the Time Warp paradigm.

The functionality of the rollback chip has been described in detail, and possible optimizations are suggested. Based on the experience with the rollback chip, the following design recommendations are suggested:

1. A write-through policy should be used. Using copy-back adds a significant amount of complexity to the design, and offers only a marginal performance advantage (if any).

2. The MRV frame number (rather than a block of written bits) should be stored in each cache entry. The latter again adds complexity to the design that is not justified by the expected improvement in performance.

3. Guidelines used in conventional cache design should be used to determine the RB cache organization (directed mapped, set associative, or fully associative) and line length.

4. Although use of the LastWA variable to indicate the latest written working area may reduce the search distance on some misses, the necessity of this optimization is questionable, especially if mark frame stacks do not grow to very large sizes.

5. The RBHistory chip need only buffer a few entries to reduce the time to update it on rollback. Eliminating the RBHistory chip completely and implementing the update operation completely in microcode is not an unreasonable alternative.

6. Fossil collection should be performed in parallel with other RBC operations to enhance performance.

In addition to the above results, other important aspects used in the RBC include the RBHistory mechanism and the use of virtual memory to avoid the need for excessive amounts of memory.

Simulation experiments were performed across a wide range of rollback scenarios, and performance data were collected and analyzed. These results are encouraging, indicating that the overall cost of implementing state saving and rollback using the RBC is only a few percent in performance. These results apply even when state saving and rollback occur frequently and version controlled memory is large.

In short, the rollback chip allows parallel programs to exploit the advantages of optimistic concurrency control algorithms using Time Warp while avoiding the overheads associated with state saving and rollback.

# REFERENCES


[1] Aahlad, Y., and Browne, J. C. Balanced Protocols for Sequencing Distributed Computations. Tech. Rep. TR-87-39, Computer Science Dept., University of Texas at Austin, Aug. 1987.

[2] Badal, D. Z. The Distributed Deadlock Detection Algorithm. *ACM Trans. on Computer Systems 4*, 4 (November 1986), 320–337.

[3] Chandak, A., and Browne, J. C. Vectorization of Discrete Event Simulation. *Proceedings of the 1983 International Conference on Parallel Processing* (August 1983), 359–361.

[4] Chandy, K. M., and Misra, J. Conditional Knowledge as a Basis for Distributed Simulation. Tech. Rep. 5251:TR:87, Computer Science Dept., California Institute of Technology, 1988.

[5] Chandy, K. M., and Misra, J. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Trans. on Software Engineering SE-5*, 5 (September 1979), 440–452.

[6] Chandy, K. M., and Misra, J. Termination Detection of Diffusing Computations in Communucating Sequential Processes. *ACM Trans. on Programming Languages and Systems 4*, 1 (January 1982), 37–43.

[7] Comfort, J. C. The Simulation of a Master-Slave Event Set Processor. *Simulation 42*, 3 (March 1984), 117–124.

[8] Denneau, M. M. The Yorktown Simulation Engine: Architecture and Hardware Description. In *Proc. 19th Design Automation Conference* (June 1982). pp. 55–59.

[9] Dijkstra, E. W., and Scholten, C. S. Termination Detection of Diffusing Computations. *Information Processing Letters 11*, 1 (August 1980), 1–4.

[10] Feridun, A. M., and Shin, K. G. A Fault-Tolerant Multiprocessor System With Rollback Recover Capabilities. *IEEE Computer Society* (1981), 283–298.

[11] Franklin, M. A., Wann, D. F., and Wong, K. F. Parallel Machines and Algorithms for Discrete-Event Simulation. *Proceedings of the 1984 International Conference on Parallel Processing* (August 1984), 449–458.

[12] Fujimoto, R. M. Lookahead in Parallel Discrete Event Simulation. *Proceedings of the 1988 International Conference on Parallel Processing* (August 1988).

[13] Fujimoto, R. M. Private Communication. September 1988.

[14] Fujimoto, R. M., Tsai, J. J., and Gopalakrishnan, G. The Roll Back Chip: Hardware Support for Distributed Simulation Using Time Warp. Tech. Rep. UU-CS-TR-87-025, Dept of Computer Science, Univ. of Utah, Salt Lake City, UT 84112, October 1987.

[15] Heidelberger, P. Statistical Analysis of Parallel Simulations. *1986 Winter Simulation Conference Proceedings* (December 1986), 290–295.

[16] INMOS Limited. *Transputer Reference Manual.* Prentice Hall, 1988.

[17] Jefferson, D. Private communication. May 1988.

[18] Jefferson, D., and Sowizral, H. Fast Concurrent Simulation Using the Time Warp Mechanism, Part I: Local Control. Tech. Rep. N-1906-AF, RAND Corporation, December 1982.

[19] Jefferson, D. R. Virtual Time. *ACM Transactions on Programming Languages and Systems 7*, 3 (July 1985), 404–425.

[20] Lee, P. A., Ghadi, N., and Heron, K. A Recovery Cache for the PDP-11. *IEEE Trans. on Computers C-29*, 6 (June 1980), 546–549.

[21] Lee, Y. H., and Shin, K. G. Design and Evaluation of a Fault-Tolerant Multiprocessor Using Hardware Recovery Blocks. *IEEE Trans. on Computers C-33*, 2 (February 1984), 113–124.

[22] Misra, J. Distributed-Discrete Event Simulation. *ACM Computing Surveys 18*, 1 (March 1986), 39–65.

[23] Pfister, G. F. The Yorktown Simulation Engine: Introduction. In *Proc. 19th Design Automation Conference* (June 1982), pp. 51–54.

[24] Pfister, G. F., and P., K. E. Software Support for the Yorktown Simulation Engine. In *Proc. 19th Design Automation Conference* (June 1982), pp. 60–64.

[25] Reed, D. A., Malony. A. D., and McCredie, B. D. Parallel Discrete Event Simulation Using Shared Memory. *IEEE Transactions on Software Engineering 14*, 4 (April 1988), 541–553.

[26] Smith, A. Cache Memories. *ACM Computing Survey 14*, 3 (September 1982), 473–530.

[27] Takasaki, S., Sasaki, T., Nomizu, N., Koike, N., and Ohmori, K. Block-Level Hardware Logic Simulation Machine. *IEEE Trans. on Computer-Aided Design* (January 1987), 46–54.

[28] Zycad Corporation. *The Zycad Logic Evaluator: Product Description.* Zycad Corp., Roseville Mn., 1983.